NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

G7256

DATA STRUCTURES AND ALGORITHMS FOR SUPPORTING GLAD INTERFACES

by

Paul D. Grenseman

June 1988

Thesis Advisor: C.T. Wu

Approved for public release; distribution is unlimited

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)

NAME OF PERFORMING ORGANIZATION aval Postgraduate School	6b OFFICE SYMBOL *(If applicable)* Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS *(City, State, and ZIP Code)* onterey, California 93943-5000		7b. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS *(City, State, and ZIP Code)*		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

TITLE *(Include Security Classification)*
TA STRUCTURES AND ALGORITHMS FOR SUPPORTING GLAD INTERFACES

PERSONAL AUTHOR(S)
renseman, Paul D.

a. TYPE OF REPORT ster's Thesis	13b TIME COVERED FROM ______ TO ______	14. DATE OF REPORT *(Year, Month, Day)* 1988, June	15 PAGE COUNT 179

SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Graphic Interfaces; Relational Databases

ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The relational database model has become the most popular and widespread database model. Most current database systems are based upon or related to the relational model. However, the relational model is beset with significant limitations, pitfalls and deficiencies. The relational model can be substantially improved with graphical interfaces. To this end, the Graphics Language for Accessing Database (GLAD) can provide easy to use and learn graphics interfaces for the relational model. Data structures and algorithms for GLAD will be presented to extend the relational model.

DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
a. NAME OF RESPONSIBLE INDIVIDUAL Prof. C.T. Wu	22b TELEPHONE *(Include Area Code)* (408) 646-3391	22c. OFFICE SYMBOL Code 52Wq

D FORM 1473, 84 MAR   83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

U.S. Government Printing Office: 1986—606-24

Data Structures and Algorithms for Supporting GLAD Interfaces

by

Paul D. Grenseman
Captain, United States Marine Corps
B.S., United States Naval Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

ABSTRACT

The relational database model has become the most popular and widespread database model. Most current database systems are based upon or related to the relational model. However, the relational model is beset with significant limitations, pitfalls and deficiencies. The relational model can be substantially improved with graphical interfaces. To this end, the Graphics Language for Accessing Database (GLAD) can provide easy to use and learn graphics interfaces for the relational model. Data structures and algorithms for GLAD will be presented to extend the relational model.

TABLE OF CONTENTS

I. INTRODUCTION

Although considerable work has been accomplished on extending and improving data models with graphical interfaces, the graphical or pictorial approach to representing a database is far from being fully explored. The potential of this approach has not been fully realized with current database systems that use a pictorial representation of the data stored in the database. Therefore, many more approaches will be proposed, explored and implemented before graphical interfaces for database are maximized for the best possible user environment and powerful semantic extension of a basic data model. Graphical interfaces can be employed to extend the user capabilities of the entire audience of database users. In particular, the naive user will greatly benefit from graphical interfaces to a database model.

Progress with graphics interfaces has largely been confined to academic and prototype systems. Most current database systems have poor user interfaces that do not represent data in an easily understandable manner. For example, the popular relational database system represents data as tuples in tables. The user interface is a standard relational type of query language that has no graphical or pictorial features. Without good user interfaces, most database systems are beset with drawbacks, pitfalls and limitations for the database user. These limitations can be resolved by extending the capabilities of these systems with additional features and by adding easy to use and learn database interfaces.

A. GLAD WILL RESOLVE SEMANTIC LIMITATIONS

Limitations with many basic data models, such as the relational model, are widely recognized. These limitations are caused by lack of semantic capability. Implementations of inadequate semantic models can result in systems that are difficult to learn and use. When implemented, the Graphics Language for Accessing Database (GLAD), proposed in Wu [Ref. 1:pp. 1-11], will provide a user with semantic capabilities that a basic model does not have and a friendly, easy to use and learn pictorial database environment. GLAD will ultimately eliminate many of the deficiencies that make the relational model difficult for the naive database user to learn and use. The naive database user will be able to access and manipulate a database with a minimum amount of difficulty. GLAD will utilize a unique graphics object approach to eliminate the relational model's deficiencies and limitations.

B. GRAPHICS OBJECTS IN GLAD

GLAD shall utilize Graphics Objects to pictorially represent data items as simple pictorial objects. These graphics objects will represent information that is stored in the database. When dealing with a database, it is natural for the user to picture objects or entities of the database and the connections that may exist between the entities. GLAD will provide the user with simple rectangles to represent entities or database objects and lines that connect the objects. This approach will be superior to simply reviewing current representations of the schema.

1. <u>GLAD Will Extend And Improve The Relational Model</u>

GLAD shall be utilized to extend the basic relational model that was developed in the early seventies and refined with a decade of theoretical research and implementation. The relational model's wide spread acceptance, popularity and successful commercial implementation has made it today's most dynamic and widely used data model. However, the relational model has significant deficiencies and limitations that makes it a perfect candidate for extension by GLAD. The successful design and implementation efforts of systems such as DB2 and INGRES can be built upon to take advantage of work that painstakingly and previously has been accomplished. Resources can be totally allocated to developing an effective higher level interface. The higher level interface that will be provided by GLAD will be a substantial improvement over the standard query language.

C. STANDARD RELATIONAL LANGUAGES

Relational Query Languages are difficult to learn and use. Semantic difficulties associated with high level, non-procedural languages should be reduced as much as possible or totally eliminated. GLAD will provide graphical interfaces that eliminate the syntactic difficulties of standard relational models. Semantic power and capability shall be added to the basic model. GLAD will support complex objects and data types. These additional capabilities will make the database environment easier to utilize.

1. Standard Query Languages Are Unsuitable For End Users

Difficulties associated with standard relational languages makes them unsuitable as an end user language for a database. High level, non-procedural languages, such as QUEL or SQL, are designed to allow the end-user to avoid mastering the details of the embedded programming language. Thus, the user does not have to understand the syntax, semantics, data structures and other features of an embedded host language, effectively eliminating the difficulties of navigating the data through the system. The user is only required to manipulate the high level set language commands to formulate the query. However, the use of set language constructs to formulate queries is not always a trivial task. A significant amount of syntax and semantics must be mastered to formulate meaningful queries.

2. An Environment That Is Unsuitable For Naive Users

The standard query language environment is not well suited to naive database users. Learning and using a conventional query language is difficult for users with or without only minimal computer science and mathematics background. This situation is unacceptable because the difficulties associated with mastering the syntax and semantics of conventional query language handicaps a large portion of the database user populace. The standard relational environment must be improved for this large category of users. Systems that do not consider large segments of the user populace will ultimately be replaced.

D. IMPROVING THE RELATIONAL ENVIRONMENT

Considerable work has been accomplished in softening the harsh and unfriendly environment presented to the user by systems that are based on the relational model. It is difficult for an office worker, administrator or secretary to relate to data representation and presentation in a manner that is dissimilar and seemingly unrelated to the normal way that they obtain, review and process data. Human factors studies have shown systems that relate new requirements to familiar concepts are easier for workers to learn and use than systems that do not consider a user's working environment and previous experience.

1. QBE and Electronic Forms

Systems such as QBE by Zloof [Ref. 2:pp. 324-342] and Query by Electronic Forms by Miyao [Ref. 3:pp. 17-25] have largely remedied the above mentioned problem. With QBE, the user is supplied example queries that can be used to formulate his associated query. The example query serves as a template for query formulation. Although this approach is powerful as a learning aid or tool, it is limited to a finite amount of frequently used query examples. If the user wants to formulate an unusual or complex query, he may be unable to find a QBE example to serve as a template for his extraordinary requirement. The Electronic Forms System provides a type of graphical representation of a form or document that is similar to an office form that is used in the business environment. The advantage of this approach is the correlation between the paper form and the electronic display screen. The mechanism that is presented to the user for accessing data is similar to a real world paper form that exists in his office environment. The drawback is that the Electronic Forms approach is very limited in scope and must be individually tailored to the business environment. Nevertheless, these system's graphical approaches to aiding the user are a significant improvement in ease of access for the database user.

2. The Early Graphical Approaches

Many of these early graphical approaches to database are user friendly but lack powerful semantic capabilities. In developing a graphical system for database representation and access, we not only want to provide a user friendly environment but also want to add additional semantic capability to the basic relational model. In particular, the graphical system must have the semantic capability of direct representation of non-atomic data.

3. Relational Data Representation

The basic relational model is unable to directly represent certain types of data. This shortcoming is due to the relational model's view of data stored as atomic types in tables. An experienced user may be able to indirectly represent non-atomic types of data by joining successive tables of data together to ultimately represent a complex data type. Unfortunately, this technique requires substantial knowledge of database beyond the scope of what should reasonably be expected from a naive database user. Glad will provide complex data type representation of non-atomic data. The details of this type of data representation will be entirely shielded from the user. To the user point of view, a complex data type is handled the same way as an atomic data type, such as integer, real or char. In fact, this is certainly not the case. A significant effort will be required to allow these special kinds of data to be directly represented in the GLAD implementation. However, from the user view, the complex data types are accessed directly because the implementation of non-atomic data representation will make data retrieval transparent to the end-user. Therefore, GLAD will resolve a serious relational database limitation, the inability to express and represent certain kinds of non-atomic information.

4. Lack of Semantic Features

The basic relational model lacks powerful semantic features. In addition to the previously mentioned problems, the relational model has the following semantic deficiencies:

1. It is unable to represent generalized entities as composites of more specialized entities.

2. The relational model is unable to represent aggregate entities as a collection of atomic and non-atomic sub-entities.

3. It is unable to classify instances of certain types of objects.

4. The relational model can not show the associations between entities in the database.

5. GLAD Will Eliminate Relational Problems

A lack of powerful higher-level abstraction representation prevents the basic relational model from providing a powerful and flexible user interface. The basic relational system can not accurately represent and display entities of a real world environment that are composed of both complex and simple sub-entities. GLAD will eliminate all of the previously mentioned semantic limitations by supporting the following powerful abstract database concepts:

1. aggregation
2. classification
3. generalization
4. association.

6. Object Oriented Approach

Powerful abstraction concepts such as the above mentioned aggregation, classification, generalization and association are best implemented with object oriented programming. These concepts require objects to be represented as composites of other objects with specializations and generalizations of object type.

E. GLAD DEVELOPMENT

GLAD is being developed with an object oriented programming approach. The object oriented approach for database development seems to be well suited for database representation. Database objects can be designed to represent real world entities in a natural and realistic manner. The object oriented approach is well suited for supporting complex data types and powerful semantic features. Conventional programming languages separate data and program instructions. This separation severely limits the capability of a conventional high level language to make an easily understandable representation of real world data objects.

1. GLAD As An Extension of The Relational Database

GLAD will combine object oriented graphics interfaces with a standard relational database system. In essence, GLAD sits on top of the relational model and adds object oriented flavor to the standard relational system. Therefore, the combined system is neither purely object oriented nor set oriented. The new system will ultimately be the synthesis of the high level, object oriented, graphics interfaces and the basic relational database system.

Since GLAD will be built on top of a relational model, the object oriented graphics language must have the ability to interface with the relational model. Queries that are received in GLAD'S object oriented syntax must be translated into relational syntax to access data that is stored in the relational backend. The retrieved data, in relational format, must be reformatted to object oriented syntax and ultimately displayed to the user via the graphics interface.

2. The High Level Interface

GLAD provides an effective data manipulation interface for the database. The graphics objects are actually used to access the information contained in the database. This approach seems to be well suited for easy learning and use by all categories of users because GLAD uses a single coherent interaction method for both the data manipulation and the data application program interactions.

The high level, graphics interface objects provide the basis for this thesis research. This research is a portion of a project that is using ACTOR by Whitewater [Ref. 4:pp. 25-115], an object oriented programming language, to develop a user friendly interface to the relational database model. In addition, GLAD's implementors hope to provide a semantically rich data model that users directly manipulate in gathering information with a minimum amount of effort. This thesis is done in conjunction and in parallel with research on a data manipulation and data definition language for GLAD. The above mentioned areas will not explicitly be covered in this paper. References to the parallel areas will only be made in conjunction with data structures or translation of the information that is held in the object oriented interface.

F. DATA STRUCTURES AND ALGORITHMS FOR THE GRAPHICS INTERFACE

This study's primary focus is the definition of data structures to store data that is pertinent to the designated database and development of algorithms that shall be utilized to translate object oriented information into relational syntax. To this end, SQL like syntax will be used to illustrate the translations. In addition, careful consideration must be given to how data shall be stored in the object oriented paradigm. There are many possible ways to store data within the system. Data should be stored in a manner that will maximize effective and efficient translation. Of secondary importance is an analysis of the basic relational model's deficiencies and validation of the need for a graphical interface.

G. OUTLINE

This research shall present one major topic per chapter. Higher level abstractions for the graphics interfaces will be discussed in conjunction with the major focus of this thesis: data structures and algorithms for supporting the graphics interfaces. A simulator using the translation algorithm is included in Appendix A to show possible GLAD translations and Appendix B to show translated output. In addition, a brief conclusion will be offered in support of the major topics. The major topic areas of this thesis are presented in the following manner:

1. Introduction
2. The Need For An Interface And Extension
3. Object Oriented Languages For Graphics Interfaces
4. The Query Windows
5. Data Structures To Support The Interfaces
6. Algorithms For The Translation
7. Conclusion.

The above listed topics are presented from a higher level perspective to a lower level perspective to present a coherent argument for GLAD as an object oriented database. In addition, data structures and algorithms to support this interface shall be discussed in detail. The topics are presented to show the user a higher level, abstract view of the requirement and a possible scheme for the physical implementation of the interface.

1. Introduction

In the introduction, the major topics of this research were introduced. The introduction shall preview the entire thesis and lead into the need for a graphics interface.

2. The Need For An Interface And Extension

In Chapter II, the need for an interface to the relational model will be validated. Chapter II will explore the history and foundation of the relational model from Ted Codd's original research to the relational based SQL and QUEL systems that are today's industry standards. These relational systems will be shown to be inadequate for the entire audience of database users. To that end, a graphics interface to the relational model will be proposed to eliminate relational limitations and deficiencies.

3. Object Oriented Languages For Graphics Interfaces

In Chapter III, object oriented concepts will be discussed. Languages that are based on these concepts are particularly well suited for implementing graphics interfaces. The language for implementing these systems, ACTOR, will be discussed and analyzed in this chapter and subsequent chapters.

4. The Query Window

The query window will be presented in Chapter IV as the principle interface to the user. A wide variety of queries shall be presented and correlations will be shown to the SQL syntax of the relational system. In addition, correspondence between GLAD queries with non-atomic data types will be discussed.

5. Syntax For The GLAD Schema

In Chapter V, the syntax for the GLAD schema will be defined. In addition, correlations between the GLAD, extended SQL and SQL schema will be shown. The example UNIVERSITY database, presented in Wu [Ref. 1:pp. 1-10], will be used to illustrate the definition of the GLAD schema. The major objects and sub-objects of the UNIVERSITY database illustrate both complex and simple data types.

6. Data Structures To Support The Interfaces

In Chapter VI, data structures to support the GLAD interfaces will be defined. The Collection is the GLAD class that will be utilized to form all of the database objects. Various descendants of the collection class, such as KeyedCollection, Set, Array and Dictionary, will be utilized to represent entities, attributes and query collections.

7. Algorithms For The Translation

In Chapter VII, algorithms for the translation of the GLAD query collections to relational format will be presented. These algorithms will be explained in detail with example queries to show the translation scheme. In addition, a simulator for object translation was constructed to show possible translations of queries. Pascal code for the simulator is located in Appendix A with output located in Appendix B.

8. Conclusion

A brief conclusion shall reemphasize the major points of this thesis. Potential future applications of GLAD will be discussed with an emphasis on military applications. In particular, the naive user will benefit greatly from learning database concepts with GLAD.

II. THE NEED FOR AN INTERFACE AND EXTENSION

If the relational database systems were entirely suitable for the entire audience of database users there would be no need for various interfaces to and extensions of the relational model. The standard relational systems are generally adequate for trained, casual and experienced, sophisticated users. But for the naive user, it is untenable. Since many users can be categorized as naive users, system designers should consider the naive users' lack of database background and training when devising the syntax and semantics of the database system. Unfortunately, many system designs gave little consideration to the novice user.

Recent advances in computer technology have made database technology available to a wide spectrum of organizations, institutions, and businesses. The demand for database technology has drastically increased as various organizations assimilate computer technology and replace old manual, automated and batch oriented systems with sophisticated on-line database systems. Systems that require the smallest amount of overhead costs and time for training will become the new industry standards.

The history and evolution of the relational model provides insight into the limitations and deficiencies of the model. In addition, the relational model can serve as a viable foundation for building higher level, database abstractions. An analysis of the relational model shall confirm the need for a powerful, user friendly interface to the relational model. The interface should have semantic capabilities that basic systems do not possess and provide easy to use and learn data access, data definition and data manipulation features.

A. A HISTORY OF THE RELATIONAL MODEL

Database management systems have evolved from simple collections of flat files to complex data base models. These models are utilized by the sophisticated products that are today's industry standards. A significant advance in database technology has been attributed to Dr. E.F. Codd. While working at the IBM San Jose Research Laboratory, Codd wrote his classic paper, <u>A Relational Model for Large Shared Data Banks</u>. This paper, Codd [Ref.5:p. 1], was the basis for the unified set of theoretical ideas that became the relational model. While reflecting on the recent history of database development, Chris Date, a former database researcher who worked with Dr. Codd, wrote that prior to the development of the relational model, database models lacked solid principles and well-defined terminology. Dr. Codd added mathematical rigor to his database model to validate his ideas [Ref. 6:p. 99].

1. Relational Products

The relational model has received wide spread acceptance and acclaim. The simplicity and the uniformity of the relational data model has contributed to its popularity. Successful commercial relational database systems have been developed. Some of the successful relational products are listed below:

1. IBM DB2 utilizes IBM hardware and IBM MVS software.
2. IBM SQL/DS utilizes IBM hardware and IBM VM & DOS software.
3. IBM QMF serves as a front-end for DB2 or SQL\DS, and uses Fujitsu OS IV F4 software.

In fact, most of the current systems are in some way based upon the relational concept. Furthermore, many prototype and research systems are actually utilizing, based on, related to or extending the relational model. Some of these systems are as follows:

1. Interface for Semantic Information System (ISIS) is an experimental system for graphically manipulating a database. ISIS is based on a simply specified high-level semantic data model. [Ref. 7:pp. 328-342]
2. Gambit is an interactive database design tool for data structures, integrity constraints and transactions. It supports the definition of static data structures in terms of the extended relational model. [Ref. 8: pp. 399-407]
3. An Intelligent Database System for End Users (AIDE/AIDE-II) provides a user friendly and an easy to use query language called AQL. [Ref. 9:pp. 25-30; Ref. 10:pp. 34-37]
4. Graphics Language for Accessing Database (GLAD). It is a unified interface method for interacting with a database. It is currently being built on top of a SQL based relational model. [Ref. 1:pp. 1-11]

B. PITFALLS AND DRAWBACKS OF THE RELATIONAL MODEL

Even though the relational database model has changed and revolutionized database technology, it is not without pitfalls and drawbacks. A bad relational database design can have significant problems. As addressed in Korth [Ref. 11:pp. 173-181], the most serious problems are as follows:

1. The Repetition of Information.
2. The Inability to Represent Certain Information.
3. The Loss of Information.

1. Redundant and Repetitious Information

Repetition of information not only wastes valuable storage space but also complicates database update. Simple changes to a relational scheme can have a large impact on the database. The addition of new tuples to the database can cause problems. In some instances, repetitive information will be needed for the new tuple to preserve the validity of the table. These additions and updates can be costly.

2. Certain Types of Information Cannot be Represented

Inability to represent certain information can occur during the database update. A problem occurs when a tuple must be added to a table, and the table does not possess values for all of the attributes that are represented in the table. The representation of an attribute without value is not easily solved. A method for representing nothing or null values must be utilized. In addition to the difficulty of determining what symbol shall designate the null value, the system implementation of null value attributes is not a trivial task.

3. The Lossy Decomposition

Loss of information can occur when relational schemas with many attributes are decomposed into many schemes with fewer attributes. More tuples but less information are obtained because many of the new tuples contain erroneous information. When the new tuples contain erroneous information, we have actually lost information and do not have access to the information that was stored in the old tuples that were decomposed. This situation is called a lossy decomposition.

4. User Created Database

From the above mentioned problems, it is easy to encounter problems with database design. An obvious cure for the problem is not allowing database users to create their own databases. Management shall require that all database definition and creation be accomplished by database professionals of the organizational, information systems or data processing department. This cure may be acceptable for a highly structured organization with a centralized data processing environment. However, it will not suffice for an organization that has a decentralized data processing department, encourages the use of computer technology, makes personal computers available to many employees, and gives the workers a great deal of flexibility and latitude in using this technology. For this type of organization, the solution centers on giving these people the tools to accomplish good database design and definition while avoiding the common relational pitfalls. It appears that basic relational systems are totally inadequate in this regard.

C. SEMANTIC LIMITATIONS OF THE RELATIONAL MODEL

Perhaps the most serious limitation of the relational model is the inability to express complex real world relationships. The inability to handle complex relationships is defined in Tsur [Ref. 12:pp. 286-295] as semantic scantiness. In Date [Ref.6:p. 609], this limitation is discussed in detail. Date writes:

> At the time of this writing (1983), most database systems relational or otherwise really have only a very limited understanding of what the data in the database means: They typically "understand" certain simple atomic data values, and certain many-to-one relationships among those values, but very little else (any more sophisticated interpretation is left to the human user).

1. Poor Semantic Capability

The relational database model's lack of semantic capability precludes this model from completely expressing the natural relationships and mutual constraints that exist between entities in the database. [Ref. 12:pp. 286-295]

For real world entities to be expressed as complex non-atomic types, cumbersome and indirect methods must be utilized. For example, postal worker could not be directly represented in the database as a type of or member of the set of government employees. The Relational Model does not allow direct representation of these complex types of data. Only atomic types such as characters, integers and reals can be directly represented in tables.

2. Semantic Deficiencies and Real World Situations

Although the relational model is very popular and in many ways has revolutionized database technology, it not only has serious design pitfalls but also is unable to handle many real world situations. We must develop a better approach to database modeling and design that can solve the above mentioned problems.

D. ALTERNATIVES FOR IMPROVING THE RELATIONAL LIMITATIONS

There are many specific approaches that can be utilized to improve the limitations and pitfalls of the basic relational database system. However, these approaches can be classified into two groups. Database designers have the following two basic alternatives for improving the relational model:

1. The relational model could be abandoned, and a new system could be designed and based on a new model.
2. The relational model could be extended with new semantic capabilities added to existing systems.

1. The Construction of a New System

The construction of a new system based on a new model is a monumental task. Developing the architecture, commands and query language takes considerable effort and time. Putting the new system into production requires arduous debugging, testing and numerous iterations. Developing the new system could be costly and unfruitful.

2. Adding Additional Features to Existing Systems

The other alternative, adding capabilities to an existing relational system, may be a faster and more effective way to develop a database system. With this approach, capabilities could be added to systems with a solid theoretical and functional base. Much of the difficulty associated with the system design and development process would be entirely avoided. Advantages of this approach include capitalizing on existing technology and saving development costs that are associated with designing new machines, backends and components. The new system, with the additional capabilities, can be built on top of a functional relational database system. The new system shall be a combination or synthesis of the basic model and higher level interface.

E. EXTENDING THE RELATIONAL SYSTEM

Consider the two following approaches that database designers have utilized to extend the capabilities of the relational system:

1. A modified query language approach that offers enriched semantic and simplified syntax.

2. An object oriented approach that provides easy to use graphical interfaces and rich semantic features for the database user.

1. Extension Through Semantic Models

To understand the rationale for the first approach, we must examine the concept of a semantic data model. Consider the Entity-Relationship (E/R) semantic model proposed in Chen [Ref. 13:pp. 1-10]. The E/R model can conceptually be thought of as a thin layer or an extension sitting on top of the relational model. An entity is a distinguishable object of some particular type or a thing which can be distinctly identified. An E/R relationship is merely a one-to-one, many-to-one or many-to-many association between the entities.

2. Advantages and Shortcomings

The advantage of this approach is that it provides a built-in set of integrity rules to the user so he does not have to explicitly formulate certain foreign key rules. Foreign key rules are implicitly understood by the system when the user specifies the type of the relationship as many-to-one, one-to-one or many-to-many. Although these concepts are powerful, they are not well-defined and left open to a designer's interpretation.

3. A Step in the Right Direction

Chen's model has enjoyed great success as a design tool. However, a successful implementation of the E/R Model has not been accomplished. Perhaps this is due to the E/R model's lack of precisely defined terms. Nevertheless, these ideas form powerful concepts that can be used to improve the relational model by making it easier to understand. Understanding is enhanced by adding enriched semantic features and simpler syntax to the basic relational model.

F. BASIC RELATIONAL SYSTEMS ARE DIFFICULT TO LEARN AND USE

It is not surprising that many of the current relational database products are conceptually very difficult for users to learn and understand. Even with simple data base operations, a user may not understand how to formulate a query to access the database. Analysis of primitive relational constructs validates the need for a natural presentation of the data to the user in an easy to understand manner.

1. Deficiencies with Relational Based Systems

Most database users do not have adequate math and computer science backgrounds to effectively understand and efficiently use the full range of primitive relational constructs. Two of the most widely used relational query languages, SQL and QUEL, are based on relational algebra and tuple calculus. Even though the design details and implementation of the data model are shielded from the user, he will need to understand certain types of math-like constructs that are related to the representation of data that is stored as tuples in tables.

Consider the complex SQL query contained in Date [Ref. 14:pp. 76-77] and shown in Figure 1.

Select supplier names for suppliers such that there does not exist a part that they do not supply.

```
      SELECT SUPPLIER_NAME
      FROM SUPPLIER
      WHERE NOT EXISTS
           (SELECT *
            FROM PARTS
            WHERE NOT EXISTS
                 (SELECT *
                  FROM SHIPMENTS
                  WHERE  SUPPLIER_NO#  =  SUPPLIER.SUP-
                    PLIER_NO#
                  AND PART_NO# = PARTS.PART_NO#));
```

Figure 1. Complex SQL Query

To formulate this query the user must have extensive knowledge of many different math and computer concepts. In addition, it must be noted that the above method is not the only way to formulate the query to retrieve the supplier names. Even though the above query is the most literal interpretation of the request, there are numerous ways to formulate this complex query. This situation can be overwhelming to the inexperienced user. For example, consider the following nested query contained in Date [Ref.14:pp. 67-68], observe the differences in query constructs, and determine which query is the easiest to use for 'Get supplier names for suppliers who supply part P2':

```
1.   SELECT  SUPPLIER_NAME
     FROM    SUPPLIER
     WHERE   SUPPLIER_NO# IN
           ( SELECT SUPPLIER_NO#
             FROM SHIPMENTS
             WHERE PART_NO# = P2);
```

```
2.   SELECT  SUPPLIER_NAME
     FROM    SUPPLIER
     WHERE   SUPPLIER_NO# IN
       ( 'S1', 'S2', 'S3', 'S4');

3.   SELECT  SUPPLIER.SUPPLIER_NAME
     FROM    SUPPLIER, SHIPMENTS
     WHERE   SUPPLIER.SUPPLIER_NO# =
             SHIPMENT.SUPPLIER_NO#
     AND     SHIPMENT.PART_NO# = 'P2';
```

Which of the above queries is easiest to formulate? The first query requires the user to understand the use of nested selects. The second query requires the use of set theory and explicit knowledge of suppliers who supply part P2. The user may not have this knowledge. The third query requires use of table joins and dot notation. In addition, there are many more methods for formulating a query for suppliers that supply part P2. The user must make many decisions to formulate a SQL query. The user's choice will depend on his background and preference.

2. Too Much is Expected From the User

Perhaps too much is expected from the user in query formation. For the example queries, we have required knowledge of the following:

1. Table joins on specific attribute conditions.
2. Dot notation used with joins of a relation.
3. Order of precedence of the query operators (i.e., nested selects).
4. Syntax of the language. Terms and meanings. (I.e., from, where, select, and not.)
5. Logical operations and meanings. Existential quantifiers and negation.

3. Difficulties in Translating the Requirement

The difficulties in understanding the syntax and semantics are not the only problems the user will have in formulating a query. Perhaps the greatest problem is the translation of the query requirement from the domain of plain English to set type constructs. The human mind does easily translate a plain English requirement to set type structures because the mind simply does not reason and think in set type constructs. Humans tend to reason in both simple and complex conditional statements that are analogous to if then else type statements. Translating the requirement to set type syntax, that will obtain a range of answers from specified domains, will not generally be trivial for naive users.

4. Substantial Training Is Required

The user is required to learn and understand a significant amount of information to access the database. In summary, this approach is acceptable for sophisticated users and computer professionals, but it will undoubtedly present problems to naive users that lack formal backgrounds in math and computer science. A substantial amount of training is required to use these systems.

G. RELATIONAL QUERY LANGUAGES MUST BE IMPROVED

In Wong [Ref. 15:pp. 22-23], the following factors are discussed and given as reasons for user difficulty in learning, understanding and using the standard query languages:

1. The user is forced to remember too many things. The user is required to remember attribute names, attribute formats, record types and values. In addition, the user must remember the meaning of certain reserve words that will be used in query formulation.

2. Standard query languages support semantically poor data models. Query languages that are based upon solid mathematical principles such as tuple calculus and relational algebra are difficult for non math oriented users. A user that has not had formal education in calculus, logic, discreet mathematics and set theory will not be able to understand the finer points and implicit assumptions that exist within the relational model.

3. There is no feedback during the query process. Users generally do not formulate a correct query on their first attempt. Logical mistakes with queries can be very subtle and hidden to even sophisticated and experienced users. A query language should have features for building a complex query in a piece meal, interactive fashion. Such features would make the formulation of a difficult query, such as example one, a much easier task. With feedback on partial results, the completed query would be correct.

4. Lack of levels of detail in schemas. In a large data base hundreds of attributes may be stored. It is very difficult for the user to select relevant attributes for a query when he must review a very large potential set of attributes for query. There is no mechanism available to control the amount of detail that is presented to the user during query formulation.

5. Lack of meta-data browsing facility. The user needs a facility to browse the meta-data to obtain a general overview of the database.

Friendlier query languages must be provided to the user to access the database. Simplifying the syntax of the query is important for extending the usability of the relational model.

H. GEM AS AN EXTENSION OF THE RELATIONAL MODEL

The GEM semantic data model, Tsur [Ref. 12:p. 286], is implemented on a dedicated backend and remedies semantic scantiness by supporting features that the relational model does not provide. Zaniola and Tsur describe their DBMS as consisting of a user-friendly front-end supporting the GEM semantic model and query language under the UNIX time-sharing system. In addition, they utilize a dedicated back-end processor to provide concurrency control, recovery and support for all database transactions. A high level diagram of the model is contained in Figure 1. GEM provides the following extensions to the relational model:

1. Notions of entities with surrogates.
2. Aggregation and generalization are supported.
3. Null values and set valued attributes.
4. GEM has extended the basic QUEL language to handle the new constructs.

1. Semantic Scantiness and User Friendliness

The system designers of this GEM implementation have not only tried to remedy the semantic scantiness of the basic relational model but also have attempted to provide a friendlier query language for the user. They have capitalized on the good features of the relational model by building GEM on top of the relational model and have eliminated the relational limitations with the more powerful GEM model. Furthermore, this system also improves recovery and performance issues of the basic INGRES system.

2. Simplified Syntax of the GEM Implementation

However, the simplified syntax seems to fall short of truly improving the difficulties associated with making a relational query. This GEM extension to QUEL still seems to be unsuitable for users with little computer science or math background. The GEM queries have potentially simplified the QUEL syntax by allowing the user to directly address non-atomic data types in the query. Without this feature, at least one additional table would be required for the query. The length of the query would be increased because either nested selects or table joins to indirectly address a non-atomic data type would be required. But, even with this abstract improvement, the same types of basic problems with formulation of queries still exist. For example, consider the following example query:

```
GEM -->  range of E is EMPLOYEE
         retrieve (EMPLOYEE.NAME)
         where EMPLOYEE.DEPARTMENT is
         E.DEPARTMENT and E.NAME = "J.Black";
```

The user is required to be familiar with ranges, attributes and conditions. These constructs can provide a considerable challenge in formulating a complex query. The user must understand the logical operations. Furthermore, he must become proficient with tuple calculus to master the GEM extension. It would seem that the goal of supplying a user friendly interface to naive users is not accomplished. The syntax is still basic QUEL syntax that allows the direct expression of certain non-atomic data types. The GEM model may be untenable for unsophisticated and inexperienced users.

Many other semantic data models that have been developed to extend the relational model have similar strengths and failures of the GEM model. However, none of the models that have been implemented with the first approach, semantic extension with a simplified query language, are both powerful and friendly. These models are generally implemented with a conventional high-level language (procedural) and accessed with a set oriented (non-procedural language). These types of languages are not well suited for providing powerful features and user friendly interfaces to the relational model.

I. VALIDATION OF THE NEED FOR A BETTER INTERFACE

Systems that utilize standard query languages as an end user interface are limited in both capability and potential use. SEQUEL and QUEL are clearly not the best choices for an end user interface to the database. However, these standard query languages have potential to aid a higher level interface to data that is stored in a relational backend.

The graphics interface of GLAD will add an additional layer of higher level abstraction to the data base system. It is possible to obtain more meaningful representations of entities as database objects instead of tables of attribute names and values. Finally, the expression of complex data types as sub-objects will provide a realistic view of the data base. The mutually supporting goals of supplying both a friendly user interface and a powerful semantic extension shall be realized with GLAD.

III. OBJECT ORIENTED LANGUAGES AND GRAPHICS INTERFACES

Object oriented programming languages, such as SmallTalk-80 and ACTOR, offer an attractive alternative to the conventional approach of developing a semantic data model. A distinct advantage of the object oriented approach is an object oriented programming language's integration of active programming instructions with passive data. In a conventional, high level programming language, such as C or pascal, data is separated from the control structures, and operations are performed on the data by procedures and functions. Since object oriented languages do not separate the data and control structures, the data and instructions are integrated into a self-contained unit called an object. The objects themselves become active elements in program execution because the code to do things such as sort, divide, square root and print is actually part of the object. An object has the capability to perform operations. The object performs an operation when it receives a message. A program is executed by an object sending messages to other objects. The operation is performed by the object utilizing a method it contains or inherits from one of its ancestors. Just as we inherit traits and characteristics from our parents and ancestors, an object inherits methods from its parents and ancestors. If a method is in an object's class hierarchy or family, the object is able to utilize the method. [Ref. 4:pp. 1-50]

Object oriented languages, such as ACTOR, possess powerful graphical capabilities. These graphical capabilities can be used to present data to the user in a manner that best represents a high level abstract view of the individual data entities and the relationships that exist between these entities. Pictorial objects can be easily accessed and tailored for any specific database. Basic graphics objects of the object oriented language are general enough to be used for many different applications.

A. OBJECT ORIENTED REPRESENTATION OF THE DATABASE MODEL

Object oriented languages seem to be particularly well suited for implementing database models. The use of objects to represent real world information is clearly superior to the conventional approach of data base modeling. The conventional approach attempts to represent real world entities as tables that contain tuples of data.

1. Database Objects Modeled For Real World Entities

Real world entities, such as student, professor and employee, have the ability to perform certain tasks. Objects modeled to represent the real world student, professor and employee can be given methods to perform operations. These operations represent real world tasks. For example, suppose the chairman of the computer science department tells his secretary to sort all of her student files in ascending alphabetical order. The secretary receives the instructions and sorts the student files by utilizing a simple method that she has developed in the office. After she accomplishes the task, she informs the chairman that she has finished sorting the student files.

2. Modeling the Objects

The database objects perform in a similar manner to the real world entities that the objects represent. In addition, the chairman could send the same message to many different objects, and each of the objects could perform the request in a different way. This is similar to the chairman giving two secretaries the same instructions. Each of the secretaries perform the instructions in a different way according to the method that each of the secretaries has learned or developed. This ability is a form of polymorphism, the property of having, assuming or passing through various forms and stages. It is a powerful concept because it closely parallels the way the human mind functions and thinks. [Ref. 4:p. 33]

3. Objects That are Easy to Use and Learn

The object oriented model can be constructed for ease of use and learning. The primary objective of this approach is to eliminate user associated difficulties with conventional query languages like QUEL and SQL. Users can be given a data model that can be directly manipulated. They shall be able to gather information and access data with a minimum amount of difficulty. This approach will make database usage available to a very wide audience. [Ref. 1: pp. 1-11]

B. THE INTERFACE FOR ACCESSING DATABASE

GLAD shall be implemented with an object oriented programming language called ACTOR. This language was developed by the White-water Group, Incorporated. ACTOR possesses powerful features that will be realized by using the Microsoft Windows environment. The White Water Group claims:

> There are many benefits to this approach,...object oriented programming makes it easier to develop, change, and debug advanced programs. Actor is a complete programming environment. It uses all the power of Microsoft Windows (MS-WINDOWS) to help you organize and analyze your work. So you can see all of your work at the same time and trace the influence of one part on another as you make changes. This makes programming in Actor a fluid, natural extension of the way you think--entirely unlike conventional programming. [Ref. 4:p. 25]

1. Easy to Use and Learn

Through implementation with ACTOR, GLAD will achieve the important objective of being both easy to use and easy to learn. GLAD will be a unique approach to utilizing graphical information to represent real world entities. With the Microsoft Windows programming environment, entire objects will be as easily manipulated by the user as simple atomic data types.

2. Coherent Interaction Method

In addition, ACTOR will be utilized to provide GLAD with a coherent interaction method for data manipulation and program development. Although GLAD is not based on a specific data model, it can be used to extend a specific model's capabilities. It possesses tremendous potential for improving a current relational system's capabilities and providing the user with a friendly graphics interfaces. Other visual models are based upon specific models and are therefore limited in capability, portability and potential use. GLAD combines the best features of higher level data base abstractions from many of the other data models. In addition, GLAD provides an efficient and effective means for interacting with the database through data definition interaction, data manipulation interaction and program development interaction. ACTOR is the ideal language for developing this type of interactive environment.

IV. THE QUERY WINDOW

The most important window interface to the user is the query window. The query window is the most frequently employed user interface for accessing information stored in the database. A result window is automatically activated by the query window to show the user the information that was accessed and retrieved. Since the information is actually retrieved from a relational backend, it is necessary to understand the high level correspondence between the GLAD window interface and the relational SQL query. Therefore, the correlations between the major types of user windows to SQL queries shall be shown. The window queries are based on the sample relational queries that are contained in a pedagogical database that was utilized in Date [Ref. 14:pp. 65-90]. Accordingly, the relational syntax for the DB2 system is standard SQL and will be utilized in conjunction with the GLAD queries.

A. GLAD TO SQL

The GLAD window is able to eliminate most relational type constructs from the user query. Essentially, a translation from the GLAD object oriented query to the set oriented SQL query must be accomplished. Before the translation algorithm is discussed, an in-depth examination of possible user queries shall be conducted, and GLAD to SQL query correlations will be reviewed.

1. <u>Simple Queries</u>

A few simple examples will be utilized to explain the simple query. The simplest queries are of the following SQL form:

- SELECT designated fields
- FROM a designated table
- WHERE a designated condition is met or true.

The GLAD query window is selected by the mouse and opened for query on the display screen. Figure 2 shows the basic GLAD query window.

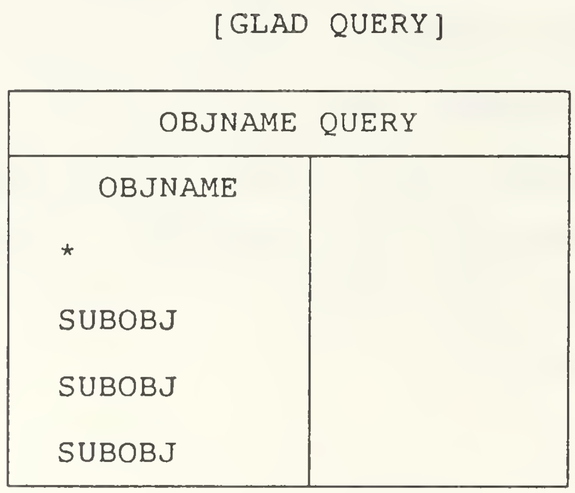


Figure 2. The GLAD General Form Query Window

After the query window is opened, instructions can be typed next to the OBJNAME, * or SUBOBJ's. The window items will be used in the following manner:

1. OBJNAME can be used for object instructions.

2. * can be used for aggregate operations and formulation of nested selects.

3. SUBOBJ can be used for retrieval of attribute values and forming conditions with relational operators.

a. Simple Query with Condition

Consider the following query:

Get supplier numbers and status for suppliers in Paris.

The GLAD query window and corresponding SQL query are shown below in Figure 3.

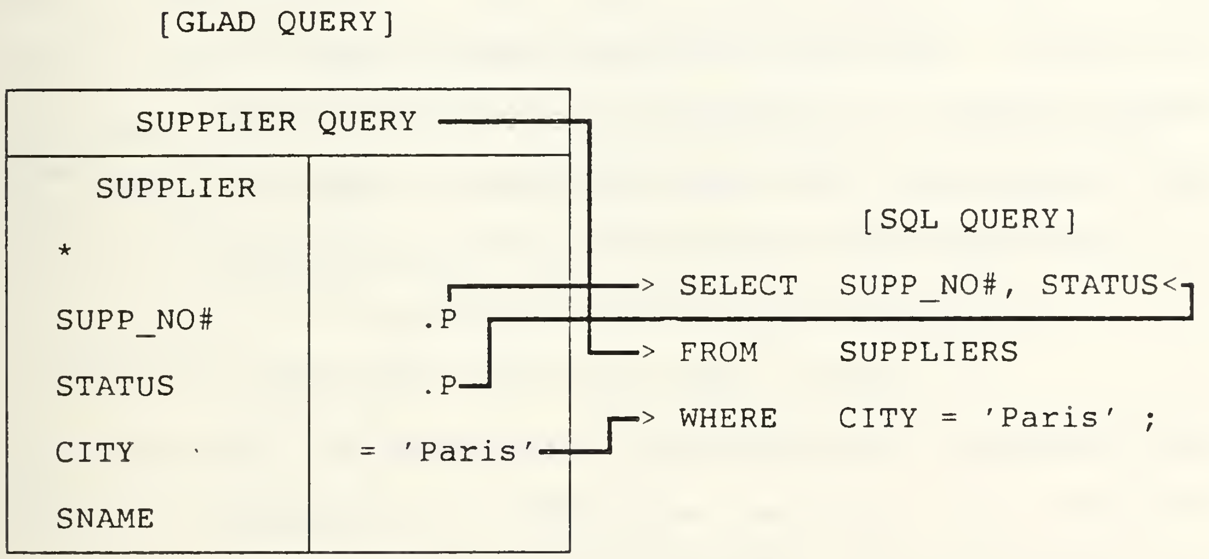


Figure 3. Simple Query with Condition

With this simple query, it is easy to correlate the GLAD query entries to the SQL query lines. The GLAD query window for the SUPPLIER object was previously selected and can be correlated to the FROM line in the SQL query. The SUPPLIER object is analogous to the SUPPLIERS table of the relational data base. Therefore, the selection of the SUPPLIERS query can automatically be substituted in the FROM line of the SQL query. The '.P', next to the attributes or sub-objects, SUPP_NO# and STATUS respectively, designates the SELECT line attributes. The user can think of '.P' as being analogous to a print command. Essentially, the system is instructed to print the sub-object values that satisfy the query. Finally, the " = 'Paris' " specifies the condition that must be met for the '.P' sub-objects to be retrieved. The CITY sub-object must be equal to the string value 'Paris'. Ultimately, attribute values for SUPP_NO# and STATUS will be retrieved from a tuple that has an attribute value of 'Paris' for the city attribute.

b. Simple Query without Condition

A simple query may be formulated without a qualification. Essentially, the system is instructed to retrieve the specified sub-object or sub-objects from a designated object. Consider the following example query:

Get the part numbers for all parts that are supplied.

The GLAD query window and corresponding SQL query are shown below in Figure 4. The user must be aware that every part number will be retrieved since no qualification is used. In addition, redundant entries in the database will not be eliminated with this type of retrieval.

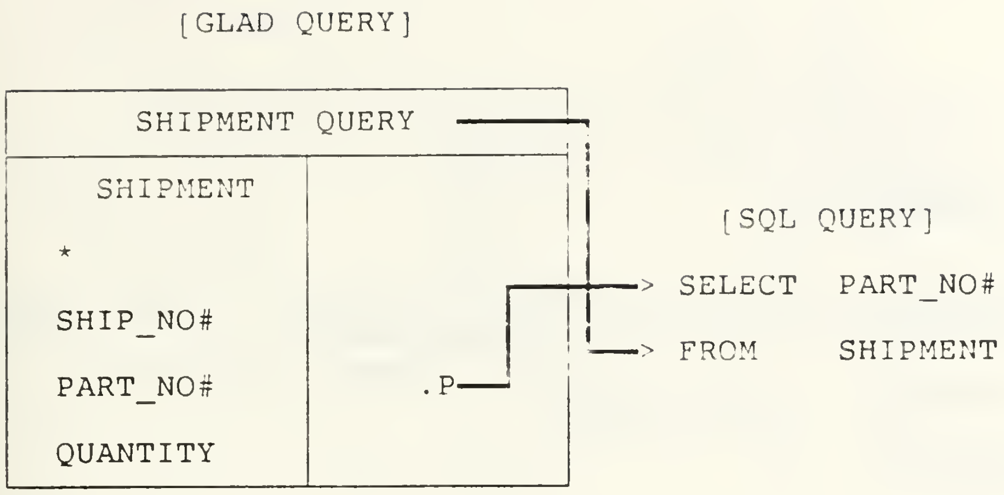


Figure 4. Simple Query without Condition

c. Retrieval with Duplicate Elimination

The above-mentioned problem can be eliminated by directing the system to eliminate duplicate items. Consider the following example query with duplicate elimination:

Get the part numbers for all parts supplied, with redundant duplicates eliminated

The GLAD query window and corresponding SQL query are shown below in Figure 5.

The redundant elements have been eliminated by specifying that the '.P' operation be done in a DISTINCT manner. Duplicate entries have been eliminated. But additional syntax, that must be understood by the user has been added to query. If eliminating the duplicates is important to the user, the trade-off of adding syntax to eliminate duplicates will be equitable.

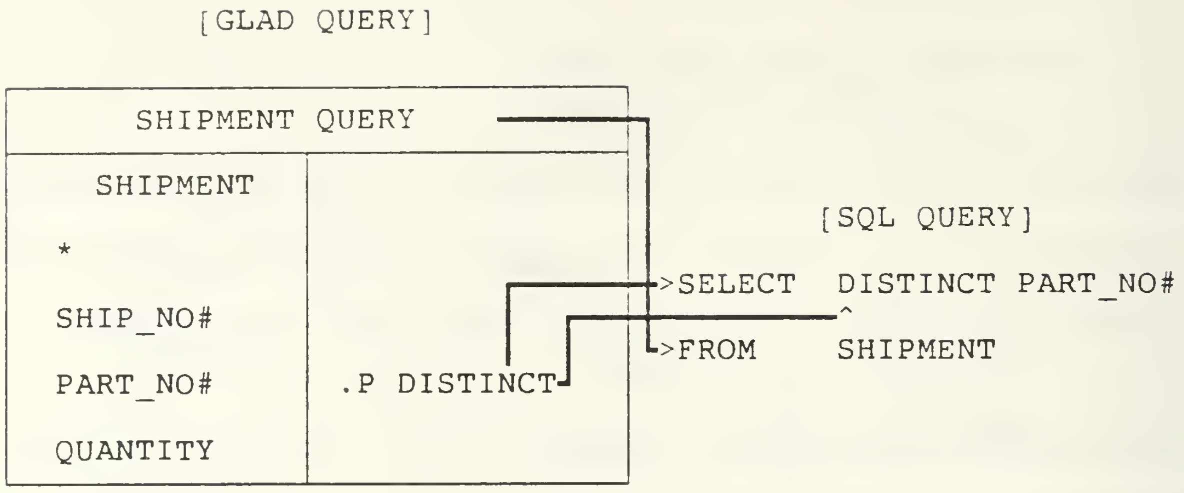


Figure 5. Simple Query with Distinct Ordering

2. Retrieval Using Computed Values

Simple arithmetic operations can be performed on selected sub-objects that have been designated for retrieval. GLAD shall allow addition, subtraction, multiplication and division to be performed on queried sub-objects to obtain certain types of numerical conversions. Conversions from pounds to grams is an example of a numerical conversion that can be accomplished.

a. Retrieval with Multiplication Operator

Retrieving a sub-object that is modified by an arithmetic operation is a simple task. Consider the following query that utilizes an addition operator:

Get the weight in grams (stored in pounds) from PARTS that are located in Paris. The GLAD and SQL queries are listed below in Figure 6.

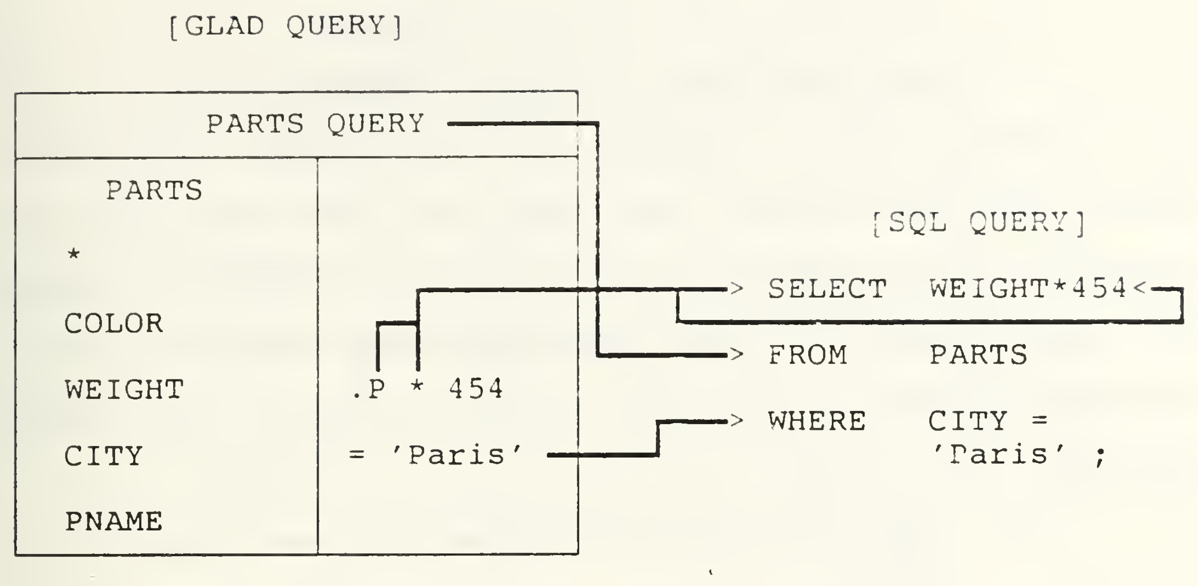


Figure 6. Retrieval with Arithmetic Operation

This arithmetic operation is easily accomplished by typing the arithmetic operator and operand immediately after the '.P' retrieval designator.

b. Retrieval with Addition Operator

Retrieval with an addition operator is handled the same way that the above multiplication operator is handled. The syntax is '.P + Number'.

c. Retrieval with Subtraction Operator

Retrieval with a subtraction operator is analogous to the retrievals that have been discussed. The user must take greater care in using the subtraction operator because the range of numbers allowable for database representation must be considered. The syntax is '.P- Number'.

d. Retrieval with Division Operator

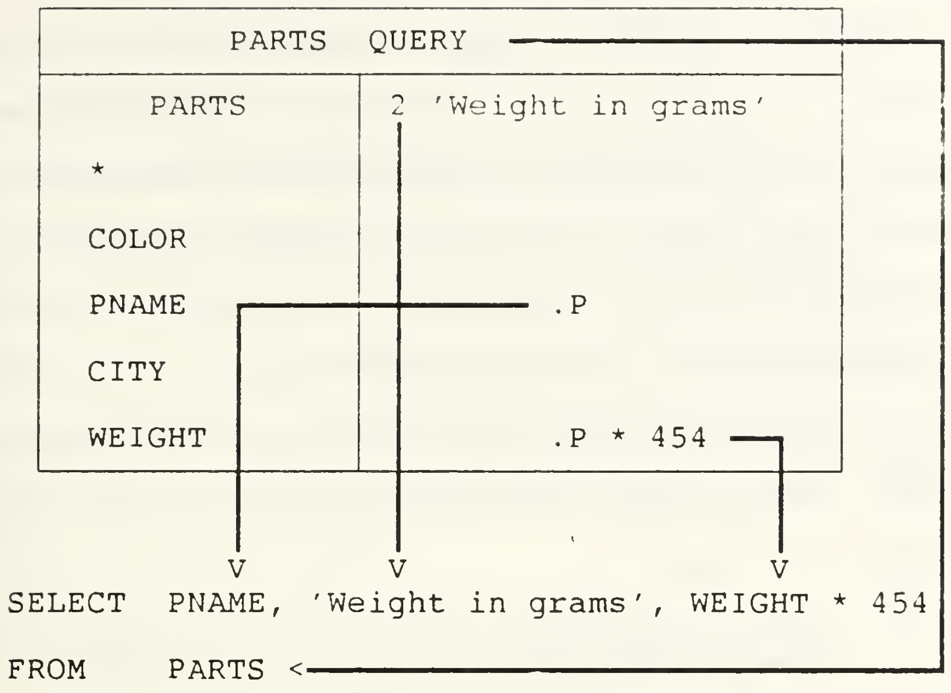
Retrieval with a division operator is similar to the multiplication, addition and subtraction retrievals. Again, the user must take care in using the division operator not only because of the range of allowable numbers but also because of gross potential error problems such as division by zero. The syntax is '.P / Number', and Number must be <> 0.

e. Use of Strings with Aggregate Operations

Additional flavor can be added to a query by specifying a message to be returned with the query results. It is possible to convert a weight that is stored in pounds to grams and describe this conversion with the returned results. Figure 7 shows how a message may be ordered. The 2 in front of the string, 'Weight in grams', indicates that the string message will be mapped to the second item in the SQL SELECT line.

It is anticipated that the results of the query will be returned to the user in the GLAD results window. Figure 8 shows a possible representation of these returned results.

The query contained in Figure 8 represents one way to use a message. Messages can be individually tailored for specific queries. They do not have to be limited to use with aggregate operations. In addition, messages can be inserted anywhere in the SELECT line by explicitly



```
SELECT  PNAME, 'Weight in grams', WEIGHT * 454
FROM    PARTS
```

Figure 7. Query with a Message

RESULTS		
PNAME	MESSAGE	WEIGHT
wrench	Weight in grams	720
bolt	Weight in grams	567

Figure 8. Results with a Message

designated position of 2, 3 or any desired position. If explicit ordering is not designated, the system translation will default to position one.

3. Print All Sub-Objects of an Object

A very desirable query feature is the ability to print out all the sub-object attribute values contained in an object with out explicitly selecting all sub-objects with '.P' operations. This ability is analogous to the SQL query where the entire table is copied.

Consider the following query:

List all the sub-object values of the SUPPLIER object.

The GLAD query and corresponding SQL query are listed below in Figure 9.

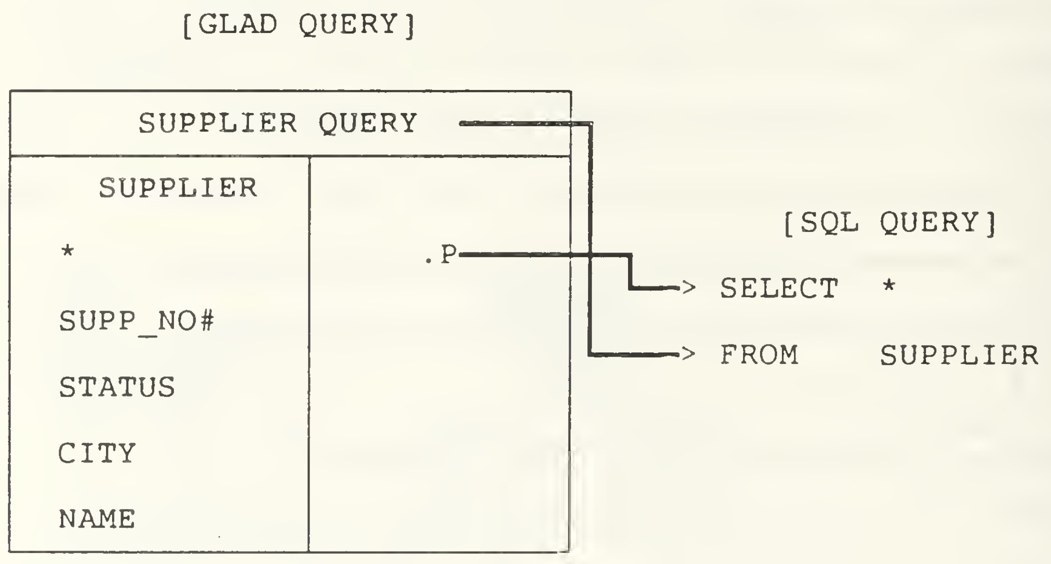


Figure 9. Copy the Entire Table

The '*' is used to represent the retrieval of all sub-objects in the designated GLAD objects. The '*' in GLAD is directly correlated to the 'SELECT *' in SQL which directs the system to produce an entire copy of the table.

Using the '*' for retrieval is a shorthand method of copying the entire table by explicitly doing a '.P' operation on every sub-object.

4. Qualified Retrieval With More Than One Condition

The user may place further restrictions or qualifications on a GLAD object by using relational operators on more than one sub-object in the query. Consider our first query, but, with an additional restriction that the status must be greater than 20. The first query is modified as follows:

> Get the supplier numbers for suppliers in Paris that have a supply status greater than twenty.

The GLAD query and corresponding SQL query show the correlation of the GLAD query to the SQL constructs. The second condition is simply represented by 'AND' which is used to form the conjunction of the conditional statements. The query in Figure 10 has more than one condition to illustrate the ability of GLAD to form conjunctions with multiple conditions.

The only difference between this retrieval and the simple retrieval is the mapping of successive conditions to 'AND' for conjunction with other SQL statements.

5. Retrieval with Ordering

The user may wish to guarantee that the retrieved information be returned in either descending or ascending order. This ordering can be explicitly specified in the query. Generally, the retrieved information is not

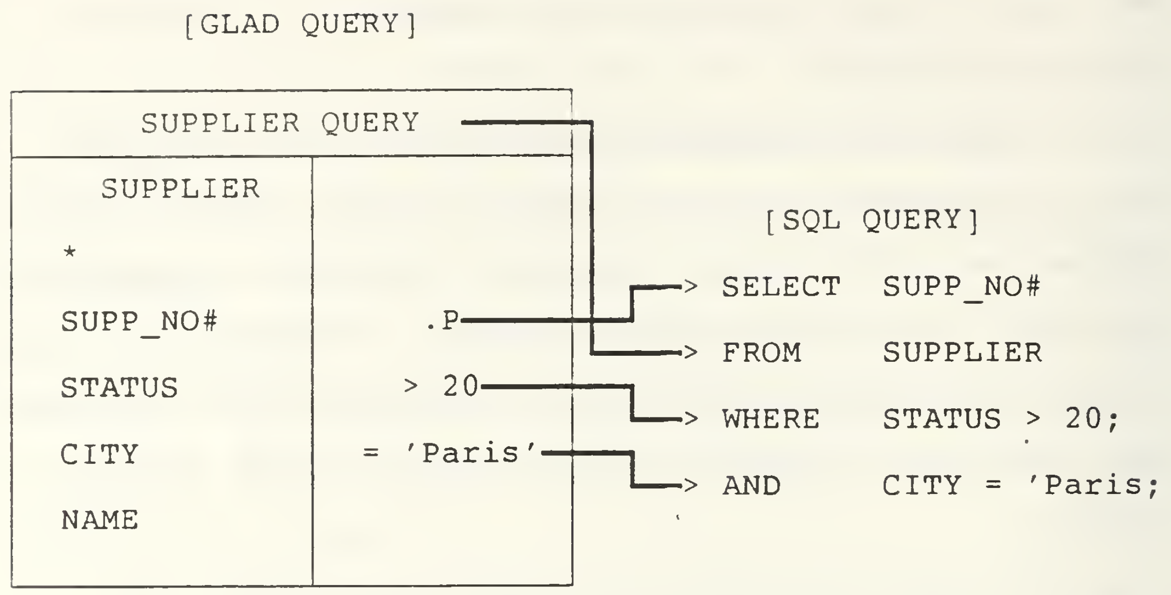


Figure 10. Retrieval with Conjunction of Conditions

guaranteed to be in any particular order. Order in either ASC (ascending order) or DESC (descending order) can be specified.

Consider the following query requirement:

Get the supplier numbers and status for suppliers in Paris, in descending order of status.

The GLAD query window and corresponding SQL query are shown in Figure 11.

The compact GLAD query must be translated to the more complicated SQL query. The '.P' and 'DESC' operation on the status attribute must be translated from the GLAD query to both the SELECT and ORDER lines in the SQL query.

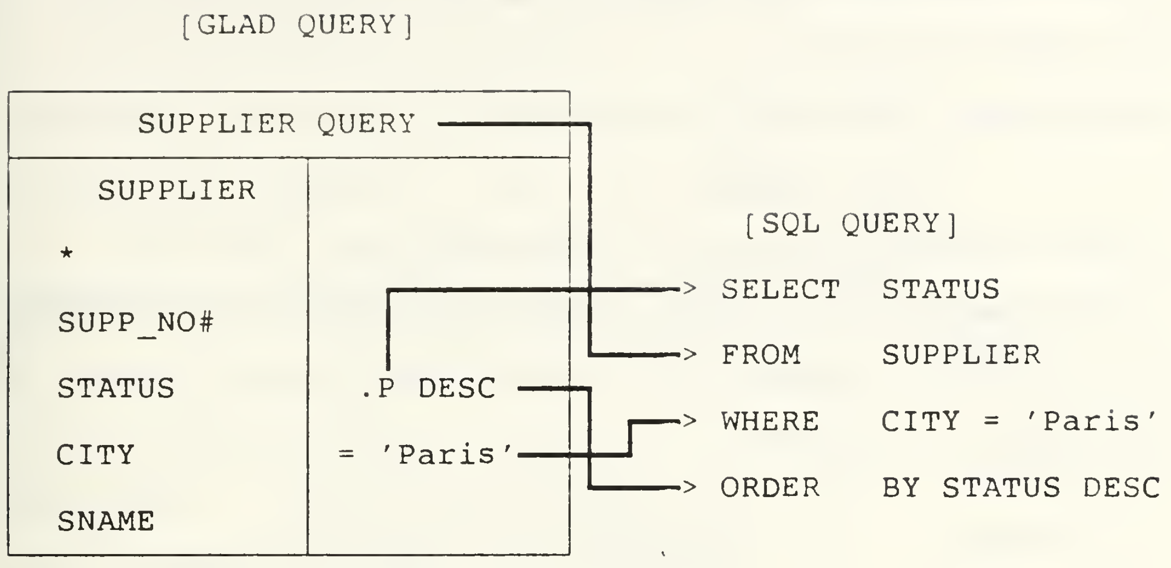


Figure 11. Retrieval in Descending Order of Status

6. <u>Retrieval Using Between</u>

GLAD will allow the use of retrieval with BETWEEN so that a user may specify an explicit range of values for retrieval of the designated sub-object. Consider the following query requirement:

Get the parts whose weight is in the range 16 to 19.

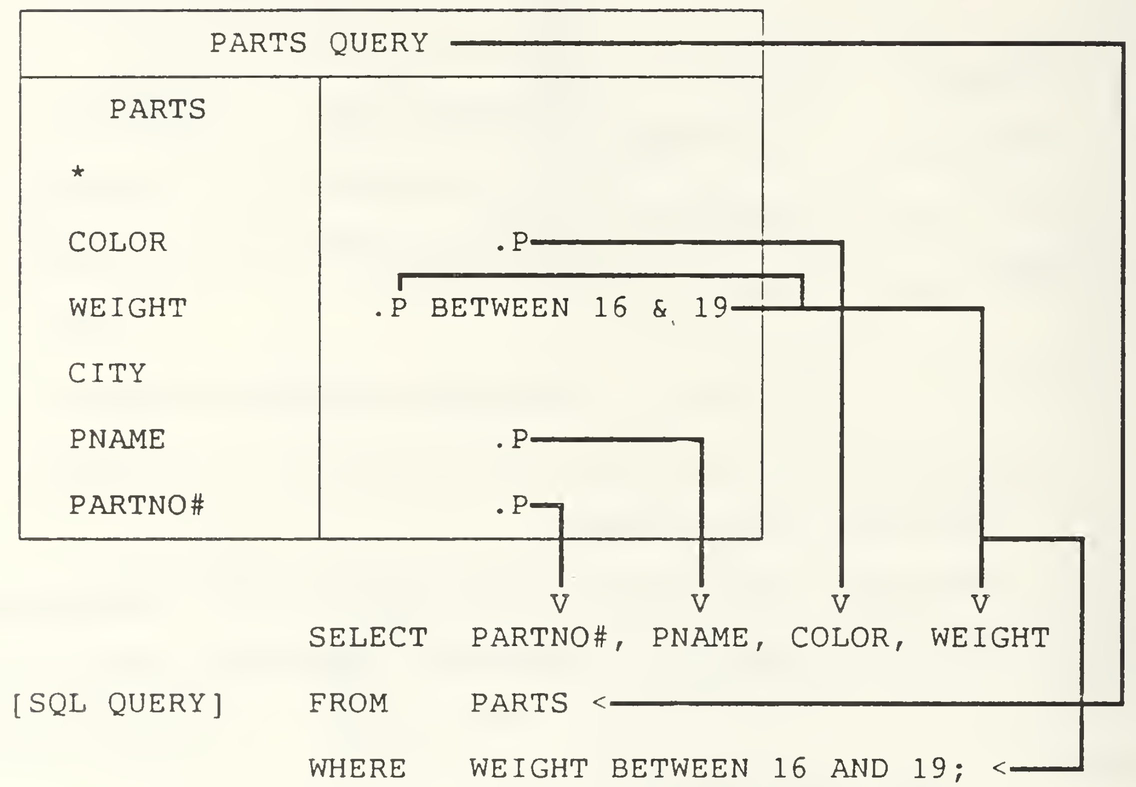
In addition to the above query, consider this query requirement:

Get the parts whose weight is not in the range 16 to 19.

The first query retrieves all parts whose weight is greater than or equal to 16 or less than or equal to 19. The second query is the contradiction of the first and retrieves values less than 16 and greater than 19.

The GLAD and SQL queries are listed below in Figure 12.

[GLAD QUERY]



```
[SQL QUERY]     SELECT   PARTNO#, PNAME, COLOR, WEIGHT
                FROM     PARTS <
                WHERE    WEIGHT BETWEEN 16 AND 19; <
```

Figure 12. Using a Range of Weight Values

The negation of the above query is accomplished simply by changing the last line to the following:

```
WHERE   WEIGHT NOT BETWEEN 16 AND 19
```

7. <u>Retrieval Using IN</u>

GLAD queries to determine if a sub-object value is an element of a user specified set of values can be stated.

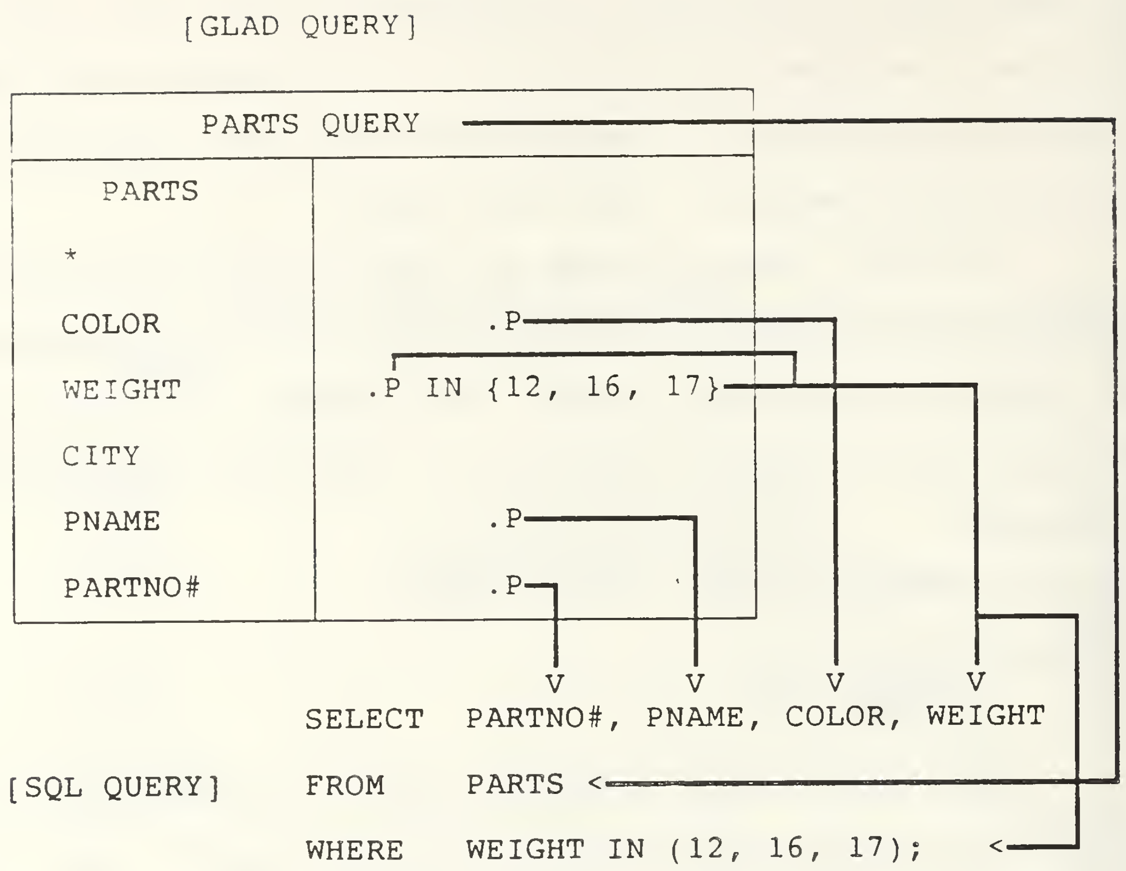
The definition of the set is dependent on the user query. This query can also be thought of as a shorthand method for a predicate involving a sequence of individual comparisons that are ORed together. [Ref. 14:p. 50]

We shall not encourage the user to utilize the OR expression when formulating GLAD queries. The OR expression can lead to subtle logic errors. For example, consider the following type of query referred to in the above referenced literature:

```
SELECT  PARTNO#, PNAME, COLOR, WEIGHT
FROM    P
WHERE   WEIGHT = 12
OR      WEIGHT = 16
OR      WEIGHT = 17;
```

Although we can explicitly allow a GLAD query to translate to this SQL format, the same results will be obtained through use of the below listed query shown in Figure 13.

The above GLAD query translates very efficiently to SQL. The syntax used in the GLAD query utilizes the familiar set type notation, { member1, member2, member3, ...}, introduced to most people in elementary school. This type of a query is an effective and efficient method of obtaining a specific result from a group of alternatives that are members of a user defined set.



```
SELECT  PARTNO#, PNAME, COLOR, WEIGHT
FROM    PARTS
WHERE   WEIGHT IN (12, 16, 17);
```

Figure 13. Retrieval Using a Set of Weight Values

8. Explicit Use of OR for Query

If the user determines that he wants to explicitly formulate a query with OR disjunctions, he can format his query in the following manner shown in Figure 14.

Essentially, the default conditions for GLAD specify that all conditional statements should be made into a conjunctive query. In order to utilize 'OR', the user must explicitly designate the use of 'OR' to prevent the translation from defaulting to 'AND'.

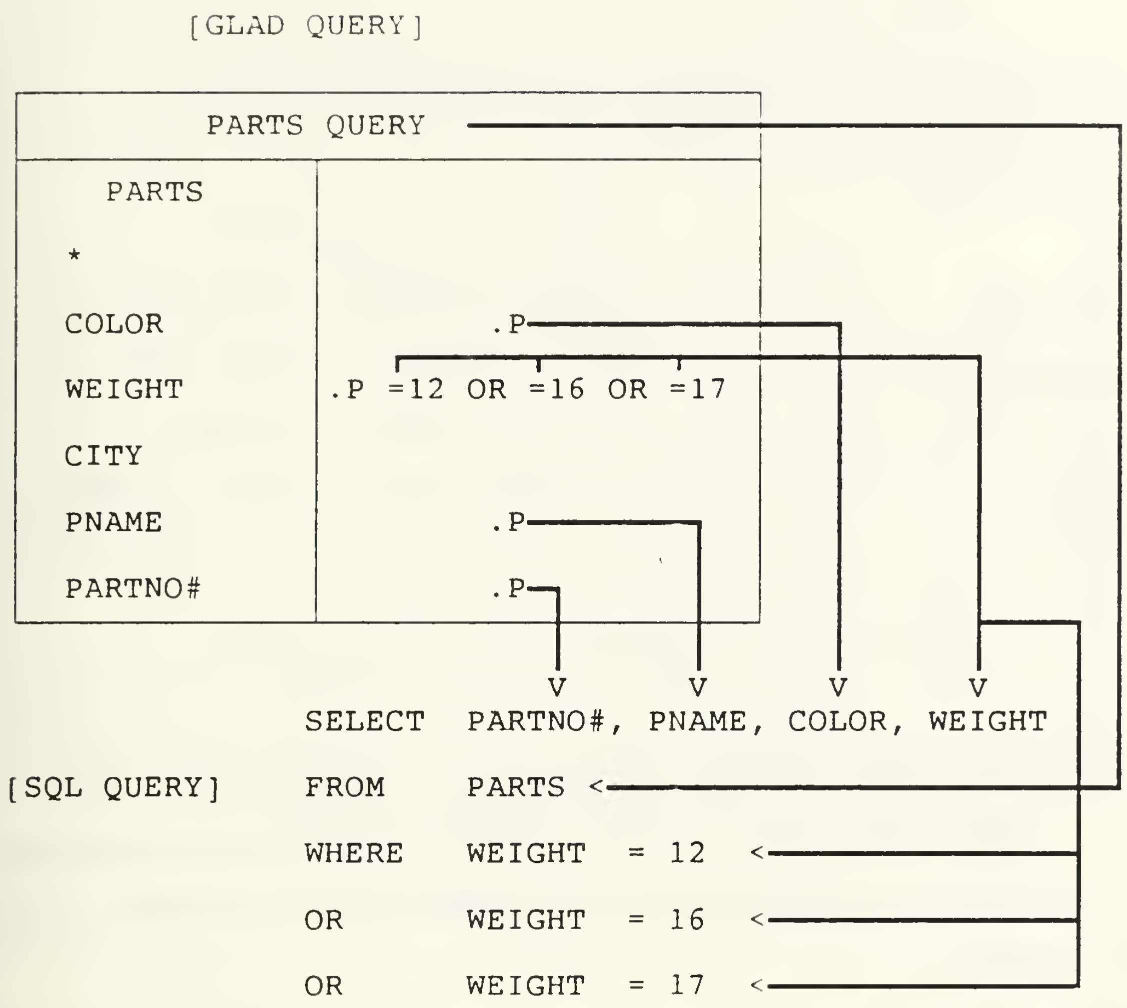


Figure 14. Retrieval Using a Set of Weight Values

9. <u>Use of OR to Form a Disjunction of Attributes</u>

A GLAD query can also be formed on the disjunction of two distinct attributes. Consider the query that was previously shown in Figure 10. Shown below is the disjunction, vice conjunction, of the attributes.

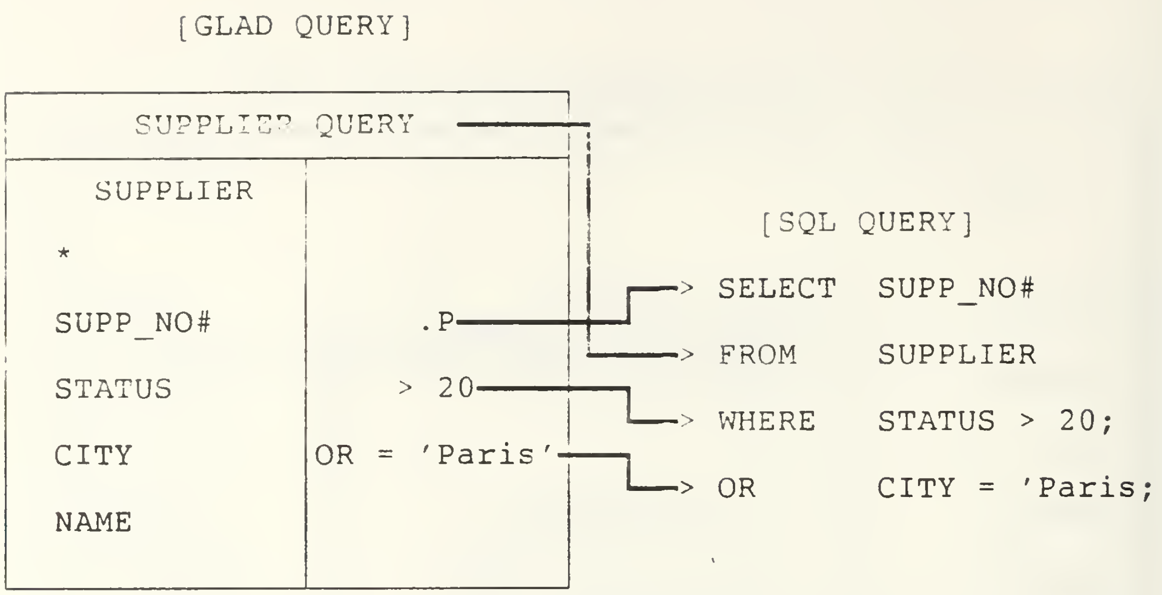


Figure 15. Disjunction of Attributes

10. <u>Retrieval Using a Character or String Sequence</u>

The user may utilize a character or string sequence to specify a retrieval. Consider the following query requirement:

Get all parts who's names begin with the letter c.

Therefore, every part name that begins with a letter c should be retrieved by this query. The length of the string or value of any subsequent characters in the string are irrelevant to the retrieval instructions.

The query displayed in Figure 16 will retrieve all the sub-object values that are contained within the PNAME that has C as the first character of the PNAME. The PNAME object is mapped to the relational PNAME attribute for retrieval, and the instructions on this sub-object are

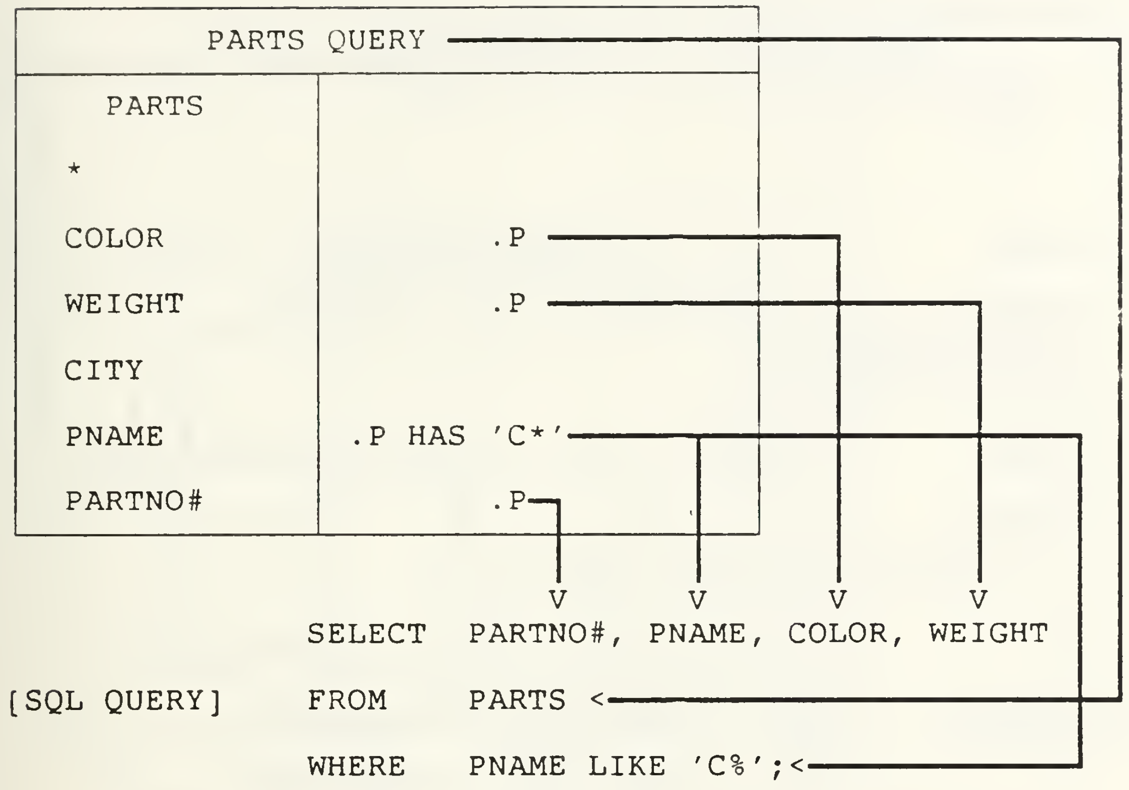
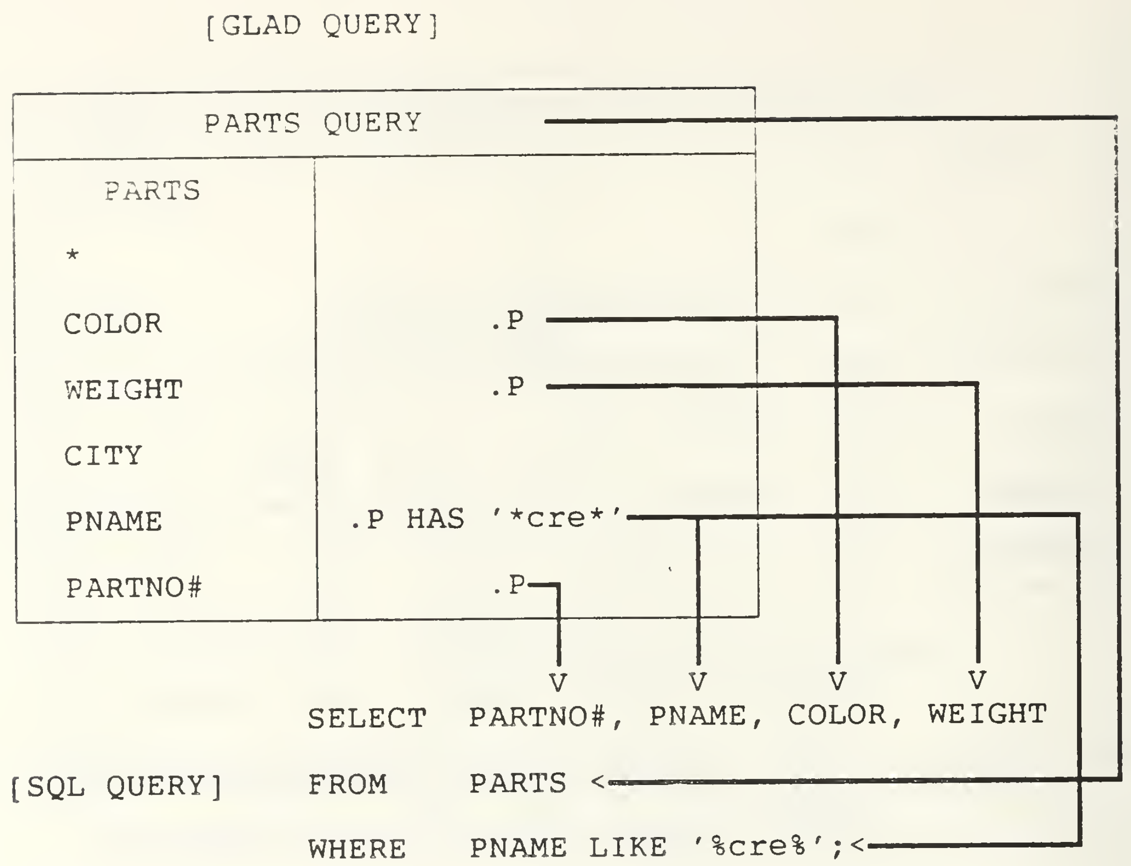


Figure 16. Retrieval Using a Search String

translated to the 'SELECT PNAME' and 'WHERE PNAME LIKE C%' portions of the SQL query.

A GLAD query can be formulated for an embedded character string in the following manner as shown in Figure 17.

This query will retrieve all the sub-object values designated for retrieval by the .P operation whose PNAME value has a 'cre' embedded in it. For example, the string 'screw' would satisfy the query condition.



```
SELECT   PARTNO#, PNAME, COLOR, WEIGHT
FROM     PARTS <
WHERE    PNAME LIKE '%cre%';<
```

Figure 17. Retrieval with Embedded Sting

11. Query with More than One Object

A powerful feature of GLAD is the ability to utilize more than one object for a query. The user can easily accomplish this task by opening more than one GLAD query window. The user may open a query window for every object in the database. Depending on how the query is formulated, the GLAD query will either translate to a SQL join or nested SELECT query.

12. <u>More than One Object--Equijoin</u>

We can retrieve all sub-objects of two tables by opening two query windows as shown in Figure 18.

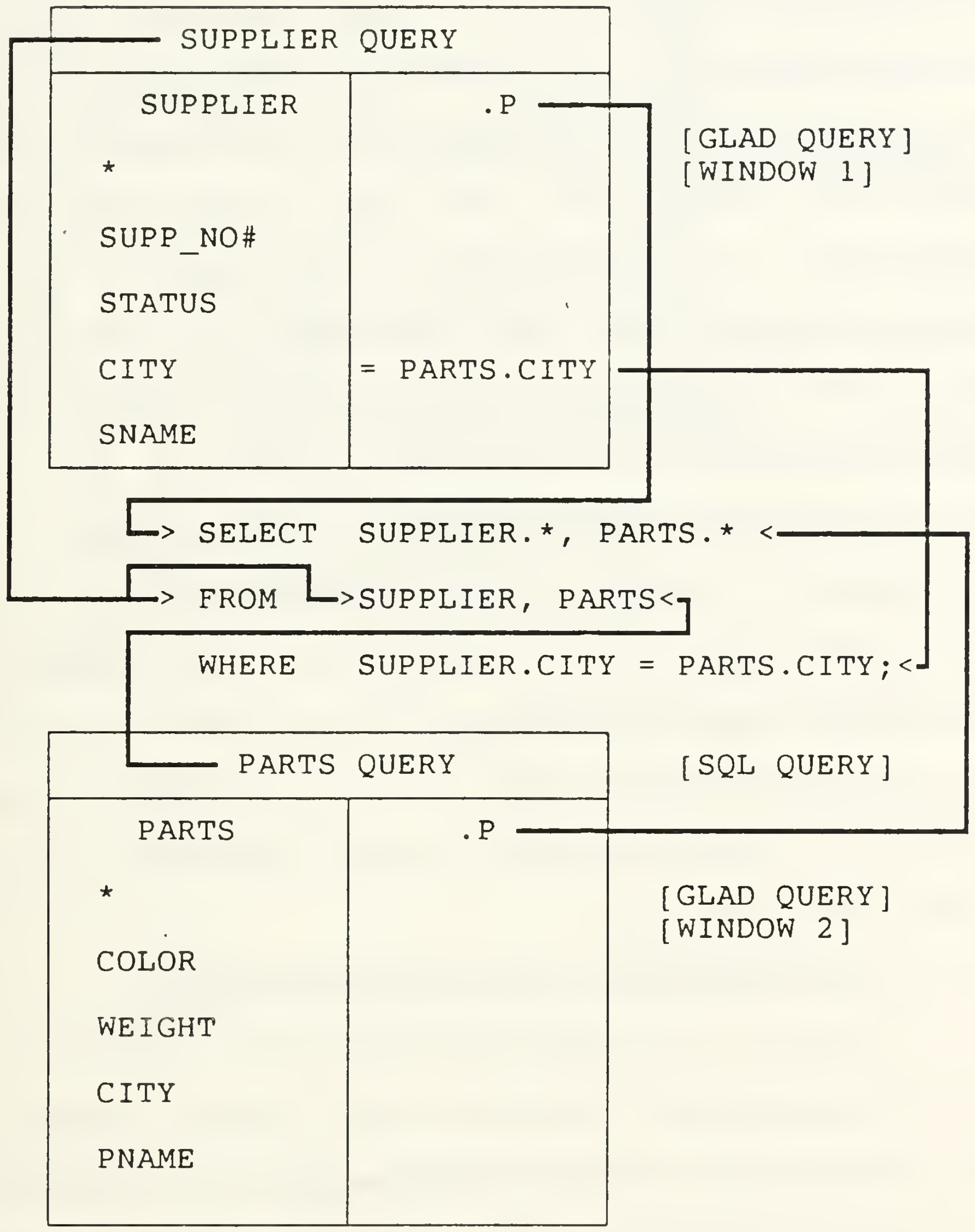


Figure 18. Retrieval with Equijoin

13. Multiple Objects with Multiple Conditions

When more than one object query window is opened, any number of qualifying conditions may be specified for the GLAD query. The first qualifier is mapped to the WHERE line and subsequent conditions are mapped to the AND line of the translated SQL query. Any ambiguity with sub-object names, such as CITY in both the PARTS and SUPPLIERS objects, is resolved by translating the GLAD query with explicit specification of object name to table name. A '.' will divide the object from the attribute, and the sub-object name will be translated to the attribute name. Figure 19 contains and example of this type of query.

14. Retrieval of Specified Fields from a Join

Specific fields can be retrieved from a multiple object query with the '.P' operations. Any number of sub-object fields may be retrieved in this manner. If ambiguity exists with sub-object names, the system will resolve the ambiguity by using the entire object name with dot notation for the translated SQL.

15. Copy Sub-Objects From Separate Tables

A copy of all sub-objects that meet specified conditions can be obtained in an efficient manner. With multiple query windows, all sub-object tuples that meet the specified conditions can be retrieved by using the shorthand notation, '.P' on the '*' for sub-object. Consider the query in Figure 19. The result window shall obtain all sub-objects

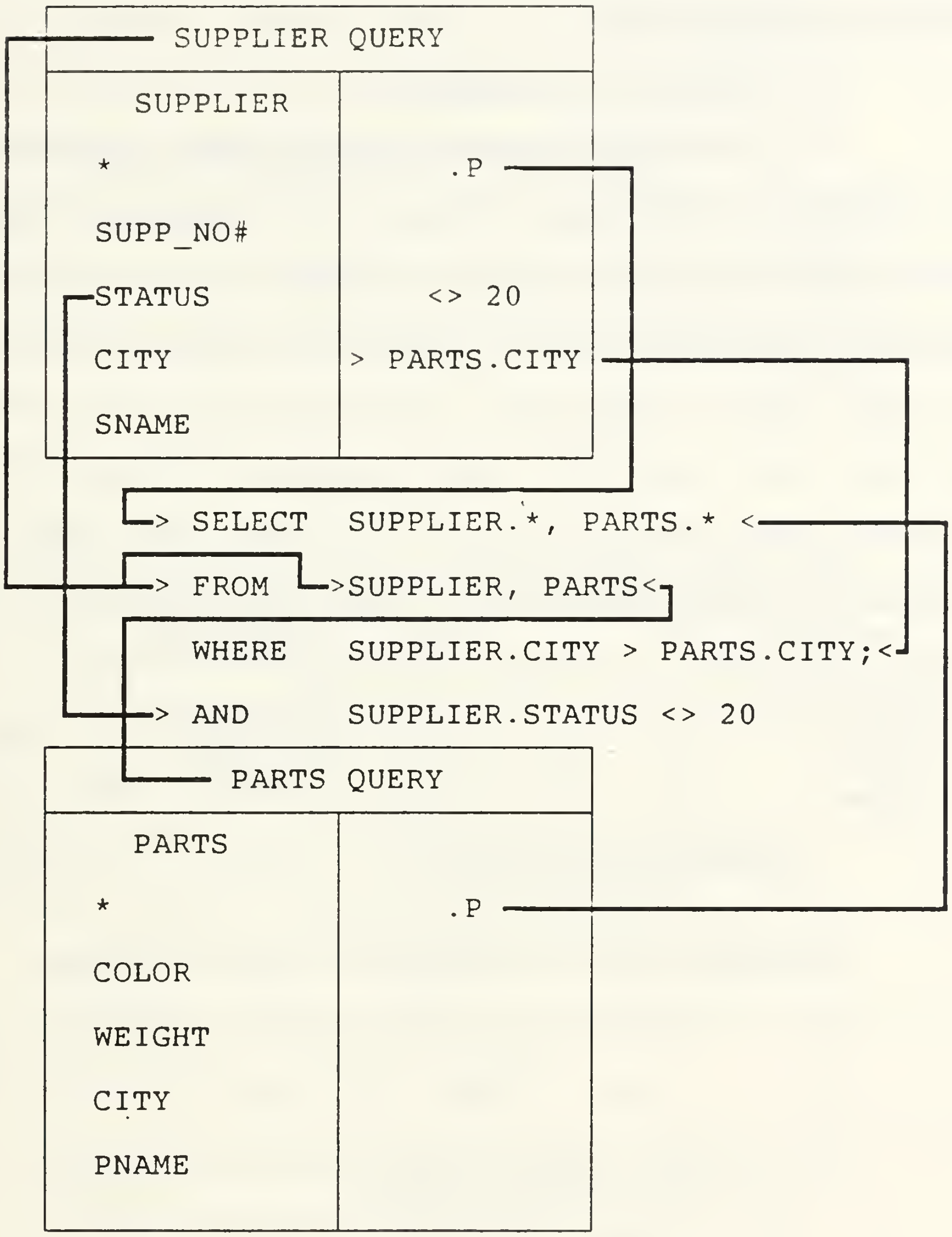


Figure 19. Copy Joined Tables on Conditions

from both tables that have (SUPPLIER.STATUS <> 20) and (SUPPLIER.CITY > PARTS.CITY). Therefore, the STATUS can be anything but 20, and SUPPLIER.CITY must alphabetically follow the PARTS.CITY.

16. Query With Three Object

As stated previously, a GLAD query is not limited to any specific number of tables. For example, three GLAD query object windows can be utilized to make a query. An example using three QUERY windows is contained in Figure 20.

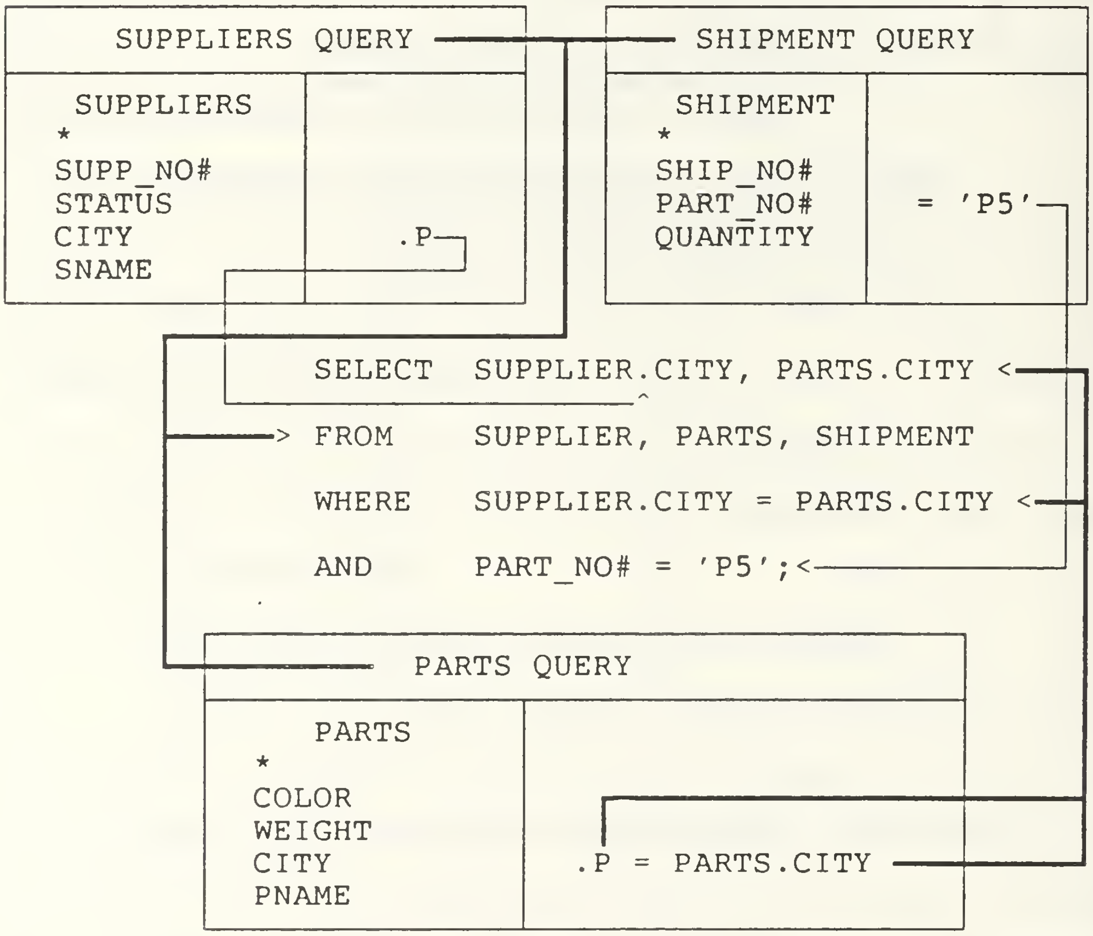


Figure 20. Three Object GLAD Query

The query contained in Figure 20 has been condensed to allow the three object windows and the correspondence to the SQL constructs on the same page.

17. <u>More Than One Object--Nested Select</u>

GLAD is not entirely limited to translating a multiple object query to a SQL query that joins tables. As mentioned earlier, the GLAD query may be translated to a SQL query that uses nested selects. Consider a query that was previously handled by simple joins. The query that gets the supplier names for suppliers who supply part 'P2' can also be obtained through the use of the existential quantifier. Consider the query contained in Figure 21. This query, that makes use of the existential quantifier, was originally expressed in the following manner:

> Get the supplier names for suppliers who supply part 'P2'.

a. <u>GLAD'S Use of the Existential Quantifier</u>

The GLAD query that utilizes nested selects is expressed as follows in plain English:

> Retrieve supplier names for suppliers such that there does exist a shipment relating them to part 'P2'.

b. <u>The Negation of the Nested Query</u>

In addition, the contradiction of the above mentioned GLAD query can easily be obtained by utilizing '.NOT EXISTS' as an operation on the '*' sub-object.

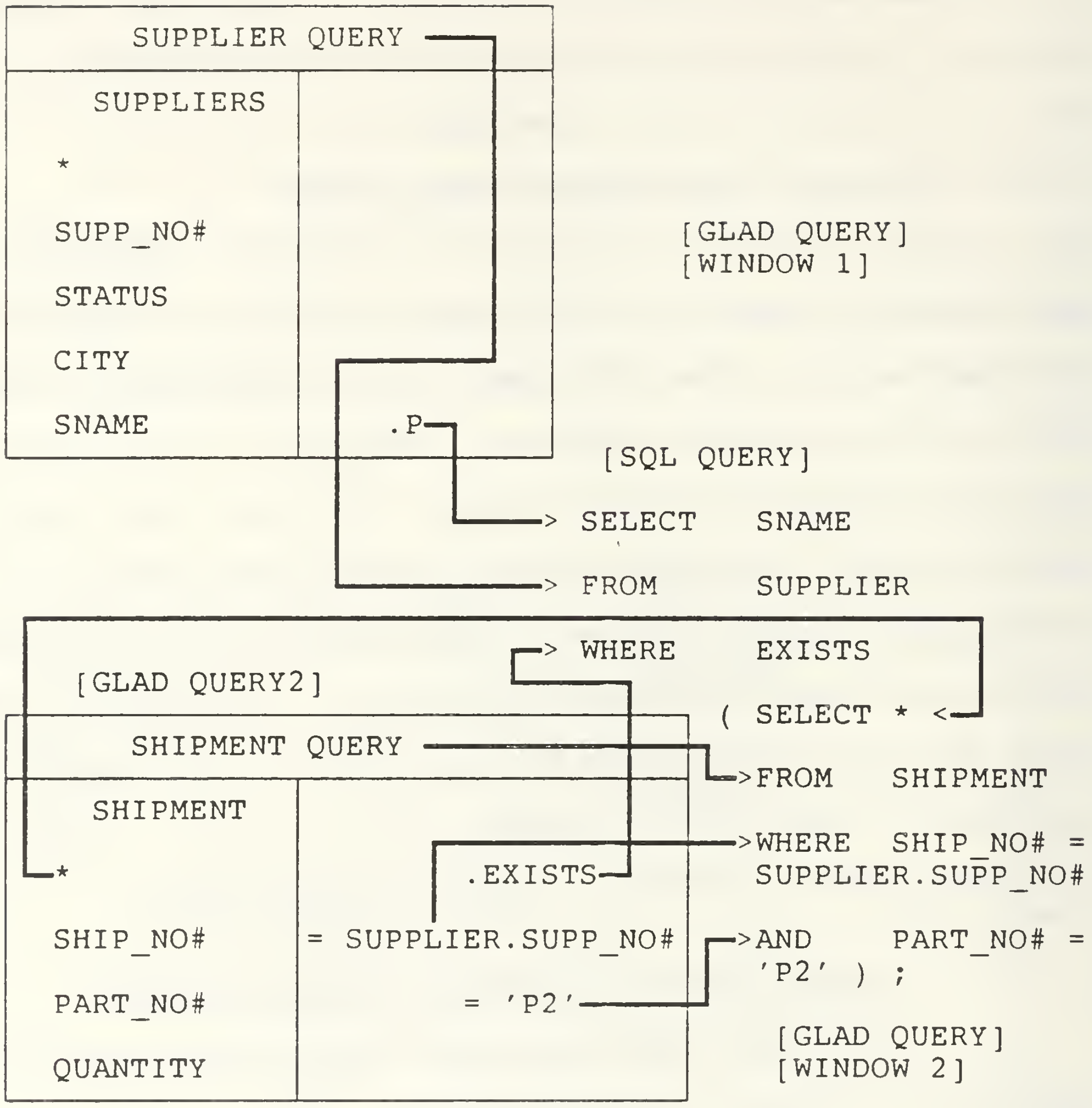


Figure 21. Query Illustrating Nested Selects

c. The Use of '*' for Translation

Finally, it must be mentioned that the '*' sub-object is displayed for every object. It is not a real sub-object that would be contained as a database object. The '*' is a special designator or pseudo object that is used for the sole purpose of translating a GLAD query to a nested select SQL query or retrieval of all sub-objects of a database object.

18. Retrieval With Aggregate Operations

Aggregate operations can be utilized with '.P' operations on sub-objects or independently on the entire object to retrieve a numerical answer that is not directly stored in the database. The query utilizes the aggregate operation to calculate the retrieved result. For example, consider the below listed queries:

1. Get the total number of suppliers supplying parts.
2. Get the number of shipments for part 'P2'.
3. Get the total quantity of part 'P2' supplied.

All of these queries use aggregate operations to calculate the retrieved results of the query.

a. GLAD QUERY--Total Number of Suppliers

Figure 22 contains a GLAD query that utilizes two aggregate operations to retrieve the number of suppliers that are currently supplying parts.

b. Retrieve the Number of Shipments for 'P2'

Aggregate Operations provide the power and flexibility to the user to obtain computed results that are not directly stored in the data base. A COUNT operation performed on the '*' in the query window will allow the user to formulate an efficient query to retrieve the total number of tuples that meet a designated condition.

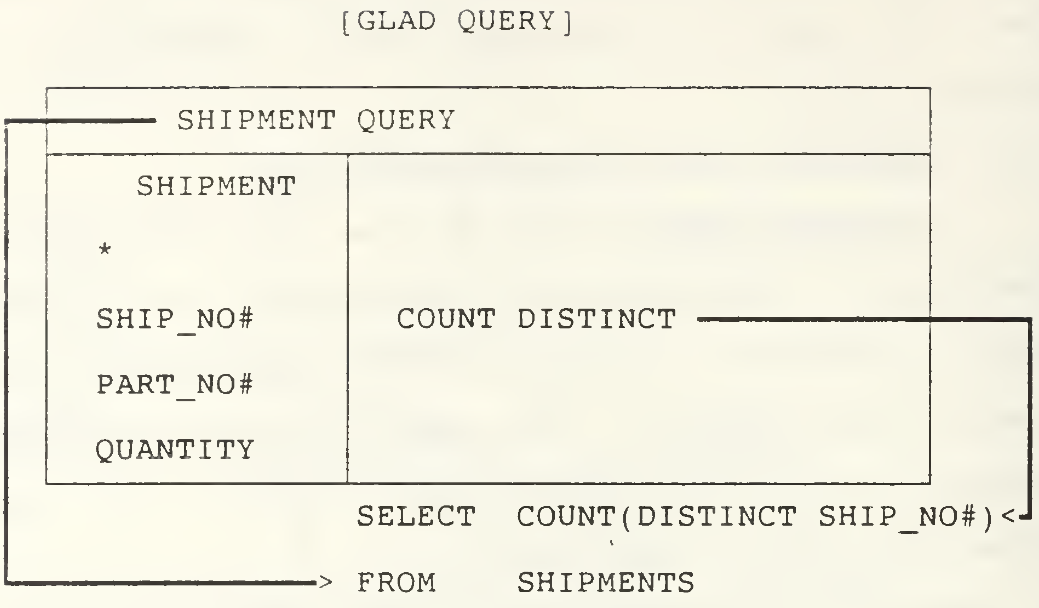


Figure 22. GLAD Query with Distinct Count

Consider a GLAD query that utilizes a simple aggregate operation to calculate the number of shipments for part 'P2'. The query is listed in Figure 23.

This query will simply count up all the shipments in the database that has 'P2' for a shipment number in the database.

c. The Total Quantity of 'P2' Supplied

Aggregate operations can be used in conjunction with '.P' operations on designated sub-objects to compute specific computed values of these sub-objects. Consider the query in Figure 24 that utilizes the SUM function to calculate the total quantity of the part 'P2' that is supplied.

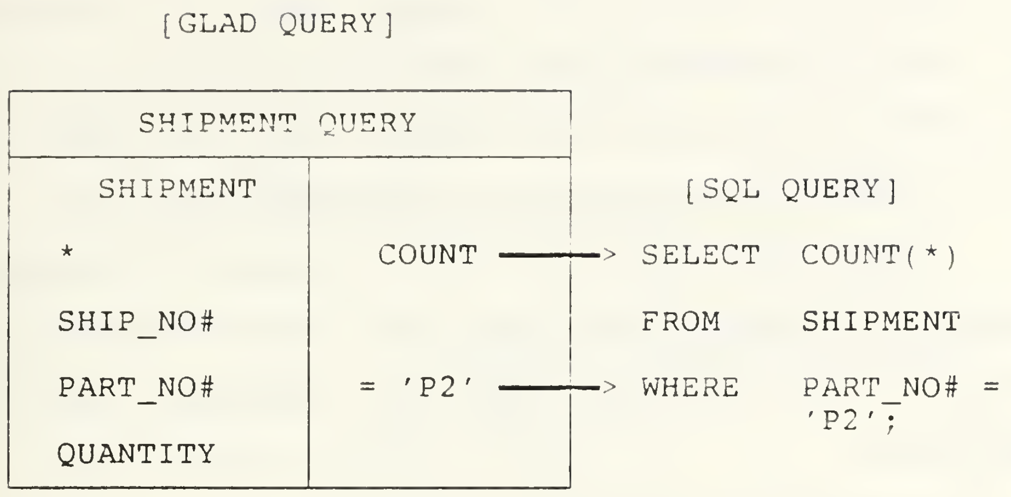


Figure 23. Using COUNT on Object Name

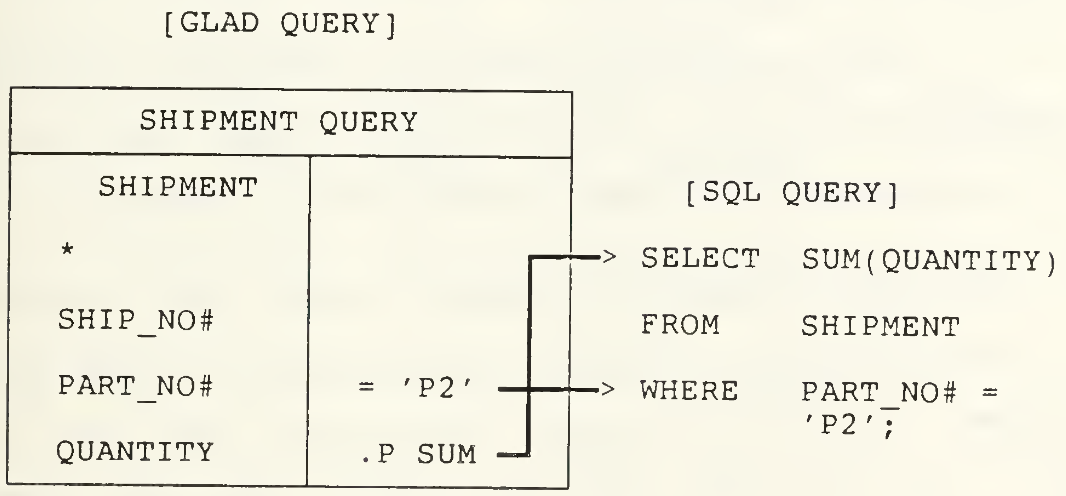


Figure 24. GLAD Query that Computes SUM of QUANTITY

In a similar manner, queries can be formulated with the following aggregate operations:

1. MAX will calculate the largest value of a sub-object.
2. MIN will calculate the smallest value of a sub-object.
3. AVG will calculate the average value of a sub-object.

19. <u>GROUP BY to Conceptually Rearrange the Data</u>

GLAD shall employ the GROUP operator to conceptually rearrange the translated table and retrieve partitions so that within one group all rows have the same value for the GROUP field.

Consider the following query:

For each part supplied, get the part number and the total quantity supplied of that part, excluding shipments from supplier 'D1'.

A GLAD query shall make use of the GROUP function to accomplish the query. Consider the following GLAD query and the corresponding SQL query translation in Figure 25.

B. GLAD QUERY OBJECTS THAT REFER TO OTHER QUERY OBJECTS

The previously discussed queries showed GLAD objects composed entirely of simple sub-objects that hold atomic data. As mentioned earlier, one of the most powerful features of GLAD is the ability to represent complex non-atomic data as simple atomic data. However, queries must be translated in a two phase process to accomplish this type of representation. First, the query must be translated to extended SQL syntax. Second, non-atomic data types must be translated to primitive SQL syntax for the final SQL query. Figure 26 illustrates the results of this process.

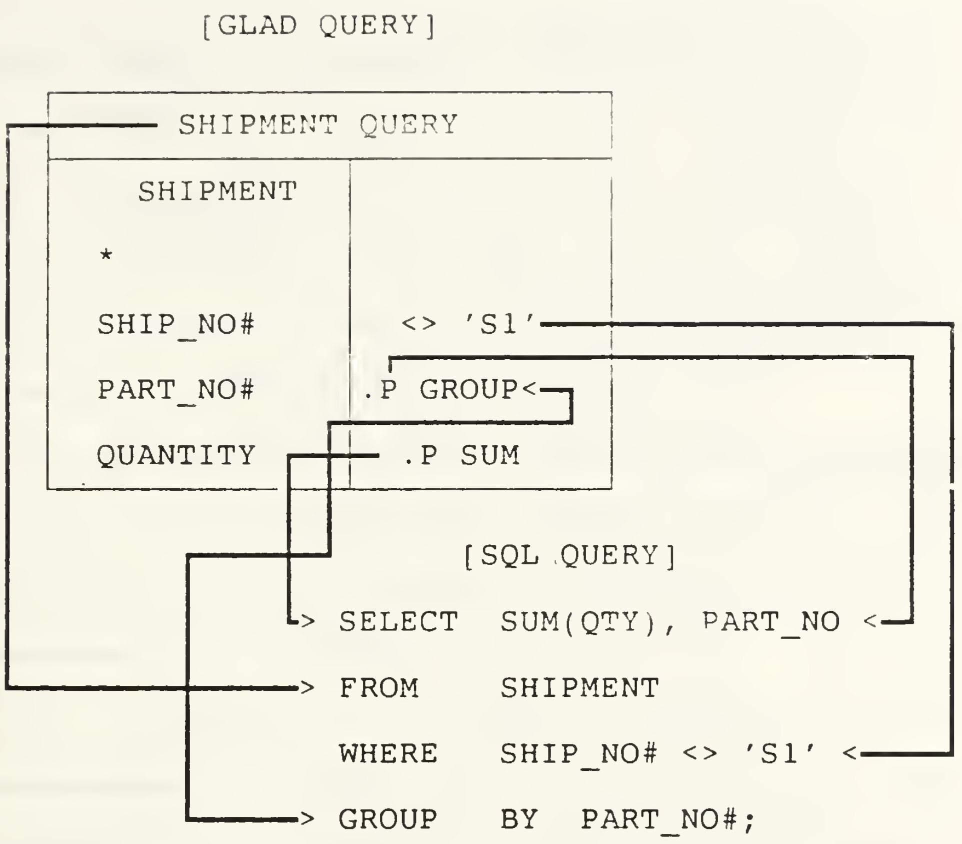


Figure 25. GLAD Query Using GROUP BY

The query in Figure 26 shows what happens when a query has a sub-object that refers to another object. In this case, the sub-object CITY is a non-atomic sub-object of type TOWN. The TOWN object consists of NAME, LOCATED and POPULATION. The TOWN sub-objects are atomic sub-objects of type integer and character.

1. Retrieval of a Non-Atomic Object

The query requires the retrieval of the SUPPLIER.CITY sub-object. Since CITY refers to another object, TOWN, all the attribute values of instances of TOWN

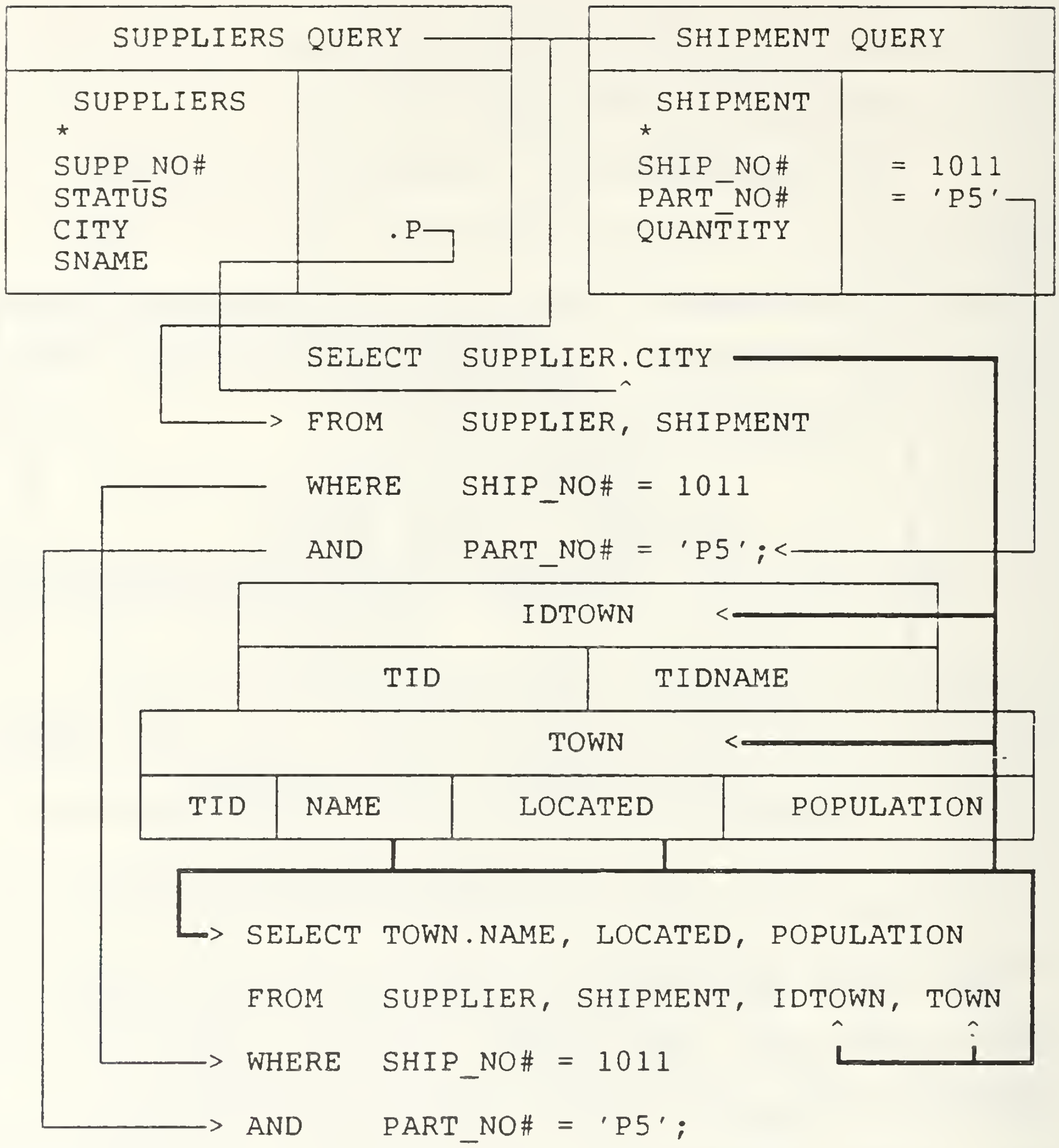


Figure 26. Query with Objects that Refers to an Object

that satisfy the query must be retrieved. Therefore, CITY is retrieved by translating the CITY sub-object to a TID (TOWN Identifer) attribute. TID is used to join the SUPPLIER and TOWN tables. SUPPLIER is joined to the IDTOWN table. The object TOWN, which is referred to by the CITY sub-object, is translated to the TOWN table and joined to the IDTOWN table. Ultimately, to retrieve the CITY sub-object that meets the query conditions, SUPPLIER must be joined to both the IDTOWN table and TOWN table. All of the attributes in the TOWN table shall be listed in the SELECT line. In other words, every tuple that satisfies the query conditions will be selected.

2. Use of Complex Objects to Specify Query Conditions

Consider the query previously presented in Figure 1. The query translation will change significantly if CITY is a complex sub-object. Consider Figure 27 with CITY as non-atomic data of type TOWN. Relational operators can be used on CITY to specify query conditions by using an identifer table that refers to the TOWN table with the complex CITY object.

In Figure 27, a query with the string value 'Paris' is used to specify a condition for the retrieval. However, CITY is of type TOWN and not type string. To avoid generating an error resulting from SQL's inability to utilize complex data, the line CITY = 'Paris' is translated to TIDNAME = 'Paris', and the SUPPLIER table is joined to an identifer table. The identifer table (IDTOWN) maintains string information that is not stored directly in the SUPPLIER table. The SUPPLIER and IDTOWN are actually joined on an integer value that is stored as an identifer in these

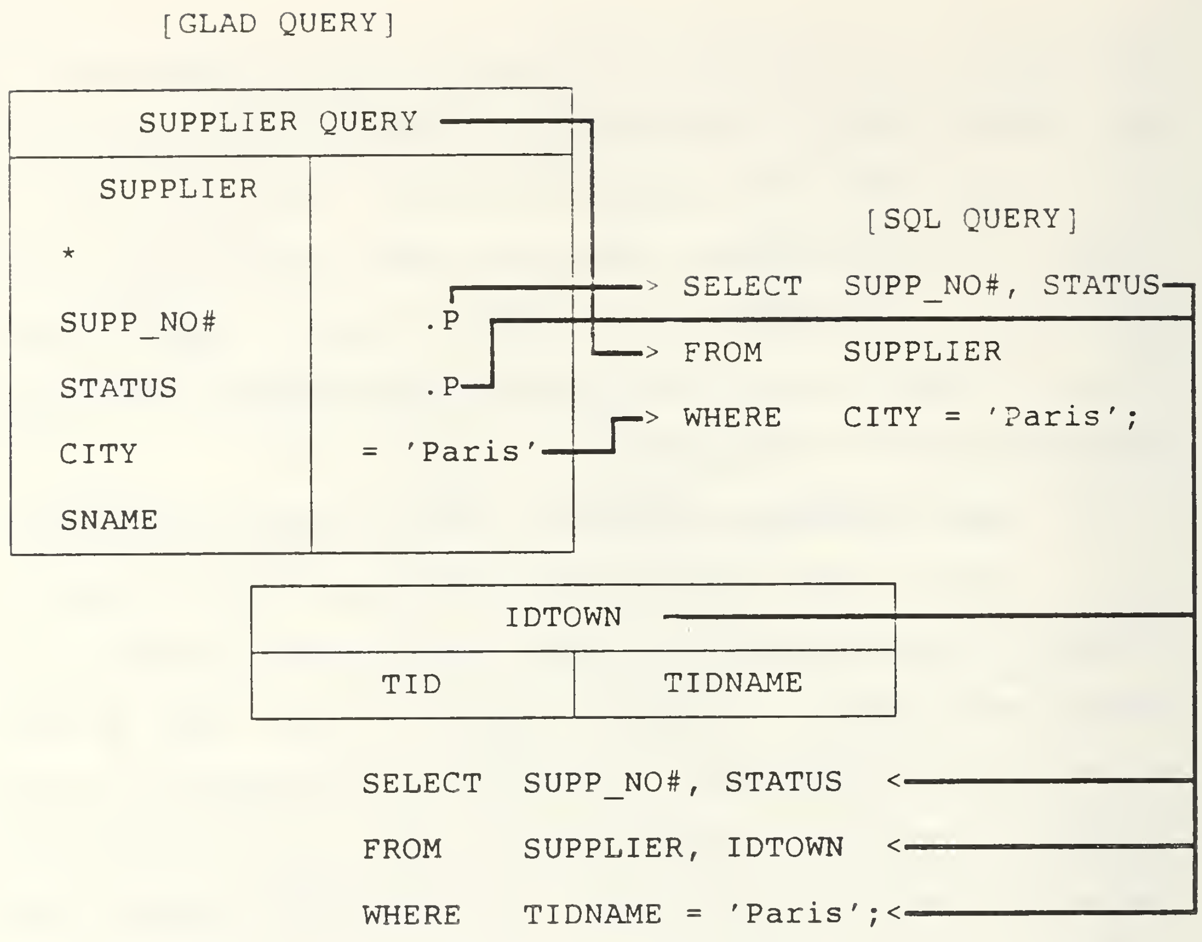


Figure 27. Non-Atomic Sub-Object with a Condition

tables. This integer, referred to as TID, is used as a surrogate value to join these tables. Since the complex sub-object is not designated for retrieval, it will not have to be joined with the TOWN table. Most conditions that utilize relational operators on complex sub-objects will be performed on the information that is held in the identifier table. For example, CITY = 'Paris' will be translated into TIDNAME = 'Paris'. The operation is actually performed on the string value 'Paris'. Therefore, unless a retrieval is specified on the complex sub-object, it will be unnecessary to join all three tables like the query in Figure 26.

C. GLAD'S CORRESPONDENCE TO SQL

The GLAD queries attempt to eliminate much of the difficulty associated with SQL queries. The GLAD queries remove much of the formulation of the relational syntax associated with SELECT, FROM, WHERE, AND and GROUP BY. Multiple objects can be used in a query by opening a query window for each object. These query windows will be translated to either a SQL join or nested select query.

Because GLAD will be built on an underlying relational model, the GLAD query windows have been developed by working backward from DB2 SQL to GLAD windows to ensure the best possible correlation. Therefore, the object oriented GLAD interface corresponds well to the SQL query language of an underlying relational system. In most cases, the GLAD query elements form a perfect one-to-one mapping to the relational backend. Therefore, at a higher, conceptual level, GLAD queries have the potential for effective and efficient transfer and translation to the relational system. Translating the data structures of the object oriented interface to a relational format requires easily accessible data structures and effective translation algorithms.

V. SYNTAX FOR THE GLAD SCHEMA

In the previous chapter, the graphical query windows to formulate a GLAD database query were discussed at the database user level. In this chapter, the higher level schema for defining objects that will support the window interfaces of the previous chapter shall be discussed in detail. The schema presented for GLAD in Wu [Ref. 16:pp. 1-10] shall be examined in detail and used with the University Database example presented in Wu [Ref. 1:pp. 1-10] to illustrate the schema definition of a GLAD database. Figure 28 contains the syntax for the schema, defined in Wu [Ref. 1:p. 3], for GLAD.

Each of the database items of Figure 28 shall be used to examine the schema for the database objects of the pedagogical UNIVERSITY database. The user interface shall provide a view of the database as graphics objects.

A. A VIEW OF THE UNIVERSITY DATABASE

Consider the high level view of the UNIVERSITY database that will be presented to the user from Wu [Ref. 1:pp. 1-10]. The UNIVERSITY database's major database objects are shown in Figure 29.

The schema definition for each of the above specific database objects shall be analyzed in detail.

```
                  Syntax For Schema Definition
                                                                  *
<db-schema> ::= <object-declaration> <object-declaration>

 <object-declaration> ::=                                   +
    DEFOBJ <object-name> <attribute-declaration>  ENDOBJ

 <attribute-declaration> ::= <attribute-name> : <type>;
                                        *
 <type> ::= <member-type> <or-list>  | SETOF <member-type>

 <member-type> ::= <system-object> | <object-name>

 <or-list> ::= OR <member-type> | OR SETOF <member-type>

 <system-object> ::= STRING [ <size> ] | NUMBER

 <size> ::= 1 | 2 | 3 | ... | maxint /* whole number */

 <object-name> ::= /* string, uppercase */

 <attribute-name> ::= /* string, uppercase */
```

Figure 28. GLAD Schema

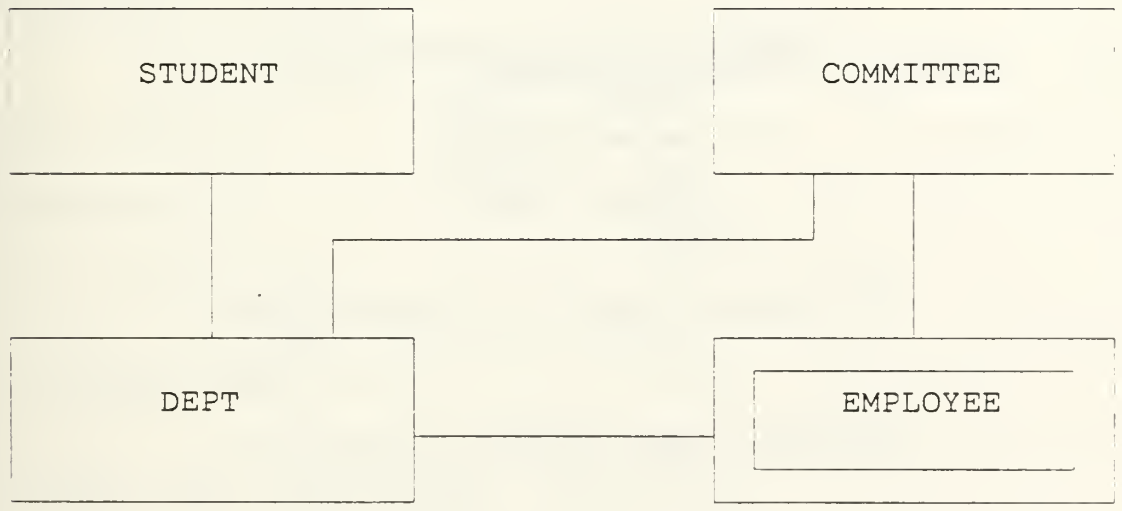


Figure 29. User View of University DB

1. The STUDENT Object Schema

The schema for the STUDENT object shall follow the GLAD schema definition convention. For the sake of clarity and space, the database definitions will be shown in a vertical manner. The STUDENT object is a complex database object that has both atomic and non-atomic attributes as sub-objects. The GLAD representation of the complex attribute MAJOR allows all of the attributes to be uniformly defined by the GLAD schema definition. Figure 30 represents the schema for the STUDENT OBJECT in GLAD, extended SQL and SQL syntax.

```
                GENERAL OBJECT DEFINITION
    <object-declaration> ::= DEFOBJ <object-name>
                                   <attribute-declaration>+
                                 ENDOBJ

               SPECIFIC STUDENT DEFINITION
     Student ::= DEFOBJ STUDENT
                    NAME: string;
                    AGE: integer;
                    GPA: integer;
                    MAJOR: DEPT;
                 ENDOBJ

             GENERALIZED ESQL & SQL DEFINITION

     <table> ::= <table-name> <attribute-declaration>+

              SPECIFIC ESQL STUDENT DEFINITION
   STUDENTTABLE ::= STUDENT (NAME:string, AGE:integer,
                  GPA:integer, MAJOR:DEPT)

              SPECIFIC SQL STUDENT DEFINITION

   STUDENTTABLE ::= STUDENT (NAME:string, AGE:integer,
              GPA:integer, DID:integer, SID:integer) &
   IDDEPTTABLE ::= IDDEPT (DID:integer, DIDNAME:string)
```

Figure 30. The STUDENT Object Schema

The schema for the STUDENT object is both easy to construct and effectively corresponds to the extended relational type schema. This extended SQL type schema is similar to the schema that was defined for the GEM extension of a relational model in Tsur [Ref. 12:pp. 1 - 8]. Since the extended SQL STUDENT table contains complex data types it is unsuitable for use as a relational query. It must be joined with an identifier table that holds a string of characters. The character string is equivalent to the name of the complex data type.

An effective correspondence between schemas shall prove to be fruitful for minimizing the translation scheme. However, we shall defer the discussion of the translation scheme and algorithms to accomplish the translation to the subsequent chapters.

Of particular importance is the treatment of the MAJOR attribute in the STUDENT sub-schema. The higher level representation of the MAJOR sub-object, which is not atomic but is represented to the user like any other atomic type, is one of the most important features of GLAD. This feature ultimately allows the user to view everything in the database as real world entities that correspond to database objects. The disadvantage is design difficulties and challenges associated with the implementation and translation of this data to primitive SQL.

The ESQL to GLAD translator can substitute an atomic ESQL attribute directly for a corresponding SQL attribute in the translated SQL query. For the non-atomic attributes, this process requires a complicated translation vice an easy substitution for atomic attributes. The translator must analyze every line of the extended query and determine if any attributes in the line are not atomic. The translator will consult a table that contains all non-atomic attributes. If an attribute is non-atomic, the translator will call procedures that will form joins on surrogate identifier tables to express the complex data in primitive SQL constructs.

2. The DEPT Object Schema

The DEPT object is related to every object in the UNIVERSITY database. The Schema for DEPT, shown in Figure 31, contains both complex and atomic data attributes.

The CHAIR object is a complex object of type FACULTY. The FACULTY object is a specialized instance of the EMPLOYEE object. NAME is a simple string type that contains the name of the university department. TENURED is a complex type of sub-object of type TPROF, and TPROF is a type of set that contains all the tenured PROFS in the UNIVERSITY database. Therefore, identifier tables will be needed for the TENURED and CHAIR sub-objects to formulate the primitive SQL query. In conclusion, the DEPT type of object is the center piece of the UNIVERSITY database.

```
                    GENERAL OBJECT DEFINITION
<object-declaration> ::= DEFOBJ <object-name>
                                   <attribute-declaration>+
                                ENDOBJ

                     SPECIFIC DEPT DEFINITION
 Dept ::= DEFOBJ DEPT
                CHAIR: FACULTY;
                NAME: string;
                TENURED: TPROF;
          ENDOBJ

                    GENERALIZED ESQL & SQL DEFINITION
 <table> ::= <table-name> <attribute-declaration>+

                     SPECIFIC ESQL DEPT DEFINITION
 DEPTTABLE ::= DEPT (CHAIR: FACULTY, NAME: string,
                 TENURED: TPROF)

                      SPECIFIC SQL DEPT DEFINITION

DEPTTABLE ::= DEPT (FID:integer, NAME:string,
               TID:integer, DID:integer) &
IDFACULTYTABLE ::= IDFACULTY (FID:integer,
              FIDNAME:string) &
IDTPROFTABLE ::= IDTPROF (TID:integer,
                  TIDNAME:string)
```

Figure 31. The DEPT Object Schema

Every object of the database is related to the DEPT object. GLAD interfaces have the unique capability of treating complex data types like atomic data types. This unique GLAD capability will be used to provide the user the best possible data definition, data access and data manipulation capabilities.

3. <u>The EMPLOYEE Object Schema</u>

The EMPLOYEE Object is a generalized object that includes FACULTY and SECRETARY specialized objects. The

Schema for the generalized EMPLOYEE object is shown in Figure 32.

```
                 GENERAL OBJECT DEFINITION
   <object-declaration> ::= DEFOBJ <object-name>
                                   <attribute-declaration>+
                                ENDOBJ

               SPECIFIC EMPLOYEE DEFINITION
    Employee ::= DEFOBJ EMPLOYEE
                    NAME: string;
                    PAY; real;
                    DEPARTMENT: DEPT;
                    JOBTYPE; CATEGORY;
                 ENDOBJ

             GENERALIZED ESQL & SQL DEFINITION
    <table> ::= <table-name> <attribute-declaration>+

             SPECIFIC ESQL EMPLOYEE DEFINITION
EMPLOYEETABLE ::= EMPLOYEE (NAME:string, PAY:real,
            JOBTYPE:CATEGORY, DEPARTMENT:DEPT)

              SPECIFIC SQL EMPLOYEE DEFINITION
EMPLOYEETABLE ::= EMPLOYEE (NAME:string, PAY:real,
          CTID:integer, DID:integer, EID:integer) &
IDCATEGORYTABLE ::= IDCATEGORY (CTID:integer,
                                   CTIDNAME:string) &
IDDEPTTABLE ::= IDDEPT (DID:integer,
                    DIDNAME:string)
```

Figure 32. The EMPLOYEE Object Schema

The EMPLOYEE object contains both complex and atomic data objects. NAME is the name of the specific EMPLOYEE. The PAY attribute is of type real to allow the decimal representation of the EMPLOYEE salary. DEPARTMENT is of type DEPT and is analogous to the MAJOR attribute of the STUDENT object. However, the user is presented with distinct conceptual representations of both of these attributes. The system must translate distinct sub-objects of the same complex type and correlate these sub-objects to relational attributes for translation. At the schema level, the translation process is accomplished with identifier tables for complex objects.

4. The COMMITTEE Object Schema

The COMMITTEE Object is a complex object that has both atomic and complex data objects. In addition, COMMITTEE is associated to the EMPLOYEE object through the MEMBERS sub-object. Figure 33 shows the entire COMMITTEE schema and the COMMITTEE to EMPLOYEE association.

```
              GENERAL COMMITTEE DEFINITION
      <object-declaration> ::= DEFOBJ <object-name>
                                   <attribute-declaration>+
                                ENDOBJ

              SPECIFIC COMMITTEE OBJECT
       Committee ::= DEFOBJ COMMITTEE
                       NAME: string;
                       MEMBERS: FACULTY;
                       PURPOSE: string;
                 ENDOBJ

            GENERALIZED ESQL & SQL DEFINITION
        <table> ::= <table-name> <attribute-declaration>+

            SPECIFIC ESQL COMMITTEE DEFINITION
       COMMITTEETABLE ::= COMMITTEE (NAME:string,
                          MEMBERS:FACULTY, PURPOSE:string);

            SPECIFIC SQL COMMITTEE DEFINITION

       COMMITTEETABLE ::= COMMITTEE (NAME:string,
                         FID:integer, PURPOSE:string) &
       IDFACULTYTABLE ::= IDFACULTY (FID:integer,
                         FIDNAME:string, CMID:string)
```

Figure 33. The COMMITTEE Object Schema

B. SCHEMA CONSIDERATIONS

An easily understandable and usable schema is important for both the designer and user of the database. The previous discussion illustrated the syntax for schema definition of the GLAD UNIVERSITY database. This higher level approach to database is well suited to naive or inexperienced users. The burden of implementing the complex sub-object types is placed entirely on the implementors of GLAD. The higher level object oriented approach to implementing the schema may be well suited for the user but presents unique challenges to the designer.

Unique design challenges await the database designer. The GLAD schema must be implemented at the physical level with effective data structures that will adequately support the user interfaces.

VI. DATA STRUCTURES TO SUPPORT THE INTERFACES

Efficient and effective data structures must be developed and employed for supporting the window interfaces to the GLAD user. GLAD's data structures maintain the information that is supplied to the GLAD window interfaces and displayed to the user. Object oriented data structures are particularly well suited for the task of supporting the GLAD windows. ACTOR has many classes that can be utilized to construct efficient and easily accessible data structures. Of these classes, various descendants of the Collection class are ideal for holding database object information. The Collection class is easily accessible through keys, indexes and elements. In addition, a Collection can store heterogenous information. Moreover, various complex and primitive data items can be stored in the same Collection. A GLAD database object is a Collection of GLAD objects. To access different databases, a global variable Collection named GLAD must be created. This Collection will hold all of the GLAD databases. In other words, we can view GLAD databases in the following manner:

1. Databases contained within the GLAD application are ultimately Collections of Collections.

2. Each database Collection consists of a group of database objects.

3. Each database object is a Collection of sub-objects that are analgous to tables and attributes of the relational model.

A. A USEFUL ANALOGY

An analogy that compares a GLAD database to an office building will be useful for illustrating the capabilites of the Collection class in developing GLAD data structures. An office building with many floors can be compared to a Collection with many elements. The Collection corresponds to the building, and the elements of the Collection correspond to the floors that are contained within the building.

The elements of the collection are database objects that define a database. Each of these objects are either related to, associated with or used in conjunction with other objects that define aggregate or generalized objects. The floors of the building correspond to the elements of the collection. Each element carries out tasks that are integrated to accomplish a function or represent specialized functions of a generalized office function. The objects of the database are collections of sub-objects. Each of these sub-objects describes the parent object. The parent object is composed of one or more sub-objects.

Rooms that are located on the floors correspond to the sub-objects of the database. These rooms carry out activities that contribute to tasks that are assigned to each floor. In other words, these room activites describe the floor's tasks.

Additional databases can be added to GLAD by increasing the size of the Collection. Collection size can be increased through the grow method. Similarly, if we need more floors for the building, additional floors can be added onto the top of the building.

B. ABOUT COLLECTIONS

The ACTOR manual states that the class Collection is the richest part of the ACTOR class tree. Collection is defined as an object that holds a group of sub-objects called elements. However, the Collection class can not be directly accessed. It is a formal class that provides universal properties for descendents. The Collection descendents serve as types of Collections that hold database objects. Array, Set and Dictionary are descendents that can be used to hold GLAD objects. [Ref. 4:pp. 169-210]

1. <u>Arrays For Glad Objects</u>

An array is a descendant of the formal class IndexedCollection. The IndexedCollection is a type of collection in which individual elements are referenced by integer values Whitewater [Ref. 4:p. 164]. In addition, the IndexedCollection unifies the Array, ByteCollection and Interval classes. ByteCollection, Array and Interval can be classified as siblings in the ACTOR class tree.

An Array can be created to hold database information. The Array information can be accessed by refering to the Array name with the bracketed index of the desired element. The Array class shall be utilized to maintain attributes\sub-objects of a particular database object. The attribute array will be one of the elements of the database object collection.

2. <u>The Dictionary Class</u>

The Dictionary is a type of KeyedCollection class. The keyed collection allows direct access to elements through use of a user specified index/key. The KeyedCollection maintains elements in an unordered manner analogous to set elements. The advantages of the KeyedCollection over an ACTOR set are substantial. An ACTOR set has elements that can only be accessed through membership operations. The element itself must be specified to be retrieved from the SET. A KeyedCollection element can be accessed through a user specified index/key. In many cases, the user may have access to the key/index but not know the name of the element. The KeyedCollection has greater flexibility than the Set and powerful operations that the Set does not possess. The below listed operations can be performed on Dictionaries that are descendants of KeyedCollections.

a. The Add Method

An element can be added to a Dictionary with the inherited KeyedCollection method Add. An Add message to the method takes two parameters and can be utilized in the following manner:

1. Datadictionary := new(Dictionary, 10); <CR>
2. add(Datadictionary,"Student", "University Object 1"); <CR>
3. Datadictionary; <CR>
4. Datadictinary("Student").

b. The At Method

The At method can be used to obtain a value that is associated with a specified user defined key. The At method can be used in the following manner:

1. at(Datadictionary, "Student"); <CR>
2. "University Object 1".

"University Object 1" is the returned item that is stored at the above specifed user defined key. If an attempt is made to access an item that either does not exist or has a non-existent key, nil is returned by the system.

c. The Remove method

The Remove method can be used to remove an item from the Dictionary. The Remove method deletes the designated element and returns the key, but if it does not find the designated element, it returns nil. Consider the following example:

1. remove(Datadictionary,"Student"); <CR>
2. "Student".

d. Other Useful Methods

The Dictionary class possesses other methods that can be used to return association objects, enumerate elements in the Dictionary, get the key and values of elements, and a host of other very useful and powerful operations.

e. GLAD Object Use of the Dictionary Class

The Dictionary will be utilized to hold the major objects of the database. Powerful methods of manipulation and access will be made available to the data user from the Dictionary Class. In addition, the database can easily be expanded through the grow method. Old objects can be purged and space obtained through the reclaim method.

The Dictionary class seems to be well suited for database objects. When information is desired about a particular object, it should be a simple matter to access the information through user defined keys. The access method is analogous to looking up information in a dictionary with the alphabetized word index.

The above-mentioned methods will be transparent to the user. The methods shall be utilized to support the window interfaces and help provide the best possible data manipulation environment. Ultimately, a Dictionary holding a GLAD database, such as the University database discussed in Wu [Ref. 1:pp. 1-10], will be created. The dictionary would contain all the major database object and would be represented by the system in the following manner:

```
  Univdb (#Student, #Employee, #Dept, #Committee)
```

3. The Set Class

An ACTOR set is a collection of unique elements. Only unique elements can belong to a Set. A Set can not contain duplicate elements, and any attempt to add an element that is already contained in the Set will fail. An element is either a member of the Set or not a member of the Set. Set membership operations are the only operations that are allowed to be performed on an ACTOR Set. An ACTOR Set can store 16K-1 elements. The cardinality of a set is maintained in the tally instance variable. [Ref. 4:p. 201]

a. Set Operations

From the Collection class, the set inherits both add and remove methods to put elements into and remove them from the Set.

b. GLAD Object Use of Sets

The Collection that maintains all of the GLAD databases shall be a set. Essentially, these databases merely have to be added to or removed from the GLAD application. Any data manipulation and data definition would be done on the specific databases. The Set class is limited in capability but is adequate to maintain the various GLAD databases.

C. GLAD DATA STRUCTURES FOR THE UNIVERSITY DATABASE

To illustrate the use of the GLAD data structures, the University Database of Wu [Ref. 1:pp. 1-10] shall be utilized to analyze database objects as instances of GLAD data objects. We shall define instances of GLAD objects for the higher-level STUDENT, COMMITTEE, EMPLOYEE and DEPT objects.

1. The DEPT Object

The DEPT object is a very important object of the UNIVERSITY Database. DEPT is related to all of the other major objects in the UNIVERSITY database. The DEPT object contains both atomic and non-atomic sub-object or attribute types. The attributes are CHAIR, shorthand for chairman, of type FACULTY; NAME, the DEPT objects only atomic attribute, of type string; and tenured, a complex set type attribute, of type TPROF. All of the other UNIVERSITY database objects contain a sub-object of type DEPT. In summary, DEPT connects the objects of the database.

a. DEPT as an Instance of GLAD Object

Consider a Dictionary named DEPT with keys of "name," "location," "members," "type" and "attributes." The keys will be utilized to access the following information:

1. "name"--The string name of the object "DEPT."
2. "location"--A point which holds the origin of the rectangle. For example, 30@30.
3. "members"--of type Deptfile that contains tuples of the DEPT object.
4. "type"--of type char. Either "a" or "g" for aggregate or generalized. DEPT is "a" since it is the aggregation of several sub-objects.
5. "attributes"--an array of attributes. Each array element shall contain an ordered pair of the data attribute name and object name. [Ref. 16:p. 6]

b. Files and Delimiters for DeptFile

In addition to the above information, the following points need to be emphasized. First, the "members" element index of the Dictionary refers to a DeptFile of type TextFile. The entries in the file will be represented in the following manner:

```
  Lum@"Computer Science"@Set("Lum" "Hsiao" "Wu" "McGhee")|
  Latta@"Math"@Set("Lucas" "Weir" "Latta" "Devito")|
```

The "@" and "|" symbols are used respectively to delineate fields and tuples. The final element of the tuple consists of a set of tenured professors for each department. The elements of this set are strings of names. The first element in the tuple is the name of the department chairman and is of type faculty. The name will probably be contained within the set of tenured professors. The name attribute is a string value. This string is not the same type as an element in the set of tenured professors, TPROF.

2. The STUDENT Object

The STUDENT object is an aggregate object composed of both atomic and complex sub-objects. NAME is an atomic attribute of type string, AGE and GPA are atomic sub-objects of type integer, and MAJOR is a complex sub-object of type DEPT. These previously mentioned objects define the aggregate STUDENT object. In essence, the complex aggregate object DEPT is nested within the STUDENT object.

a. STUDENT as an Instance of GLAD object

An instance of the STUDENT object is similar to the instance of the DEPT object. The major difference is that STUDENT maintains DEPT as one of its attributes or sub-objects. In other words, the STUDENT object is an instance of a complex GLAD object. STUDENT maintains another instance of a primary database object, MAJOR of type DEPT, as an attribute. The attribute array maintains the attribute names and shall be represented in the following manner:

```
  -  STUDENT("attributes") =
        Array( Array("NAME" #basic) Array("AGE" #basic)
        Array("GPA" #basic) Array("DEPARTMENT" #DEPT);
```

As stated in Wu [Ref. 16:pp. 1-7], basic is used when the attribute type is defined in ACTOR. The Object name is necessary for appropriate shading to describe the window interface. If the object name is not defined in ACTOR, the name of the object will be a complex type, such as DEPT or TPROF, that has been defined for the database.

3. The EMPLOYEE Object

The EMPLOYEE Object, of the University database, is a different kind of object than the DEPT and STUDENT objects that have been discussed. EMPLOYEE is a type of generalized object. This type of object is a generalized type of a more specialized object. FACULTY and SECRETARY are specialized objects of the more generalized EMPLOYEE type. The attribute "JOBTYPE" of type CATEGORY indicates the type of the specialized EMPLOYEE.

4. The COMMITTEE Object

The COMMITTEE Object is an aggregate object. It has attributes of "NAME", "MEMBERS" and "PURPOSE." The "MEMBERS" attribute is of type FACULTY. FACULTY is a complex specialized object that is nested within COMMITTEE. This implicit relationship forms an abstract, higher level association. Specifically, COMMITTEE is associated to EMPLOYEE through the specialized FACULTY object.

5. The Higher Level Abstractions

The above objects illustrate how the higher level abstractions of generalization, aggregation and association are supported by GLAD. In addition, GLAD also supports classification of objects. Every specific element of an object can be classified as that type of object. For example "John Smith" is an element of "MEMBERS" of type StudentFile. Therefore, "John Smith" is classified as a STUDENT.

D. THE QUERY COLLECTION

The data structures that collect information from the query window shall be designated as query collections. Essentially, these data structures shall take information from the database user via the query window and pass it to a translator. An array shall be utilized to obtain the query information.

1. Query Array Entries

The array entries shall contain all of the data that is pertinent to the designated query. Each entry shall consist of an ordered pair of items. The first entry shall contain the object name and any instructions to be performed on the query object. Subsequent entries will contain the attribute names and any instructions to be performed on the attributes. The following is the general form for a possible representation of the GLAD query collection:

```
-  QUERYCOLLECTION("OBJECTNAME") =
      Array( Array(OBJNAME OBJINST) Array(AttName1 Inst1)
      Array(AttName2 Inst2) Array(Attrname3 Inst3);
```

2. An Example Query Collection

The general form contains all of the pertinent information of the query collection in object oriented format. A specific example shall further illustrate the query collection. Consider a query of the STUDENT object of the UNIVERSITY database where the goal is to retrieve the names of students who have grade point averages equal to or higher than 3.5. After the user inputs information into the STUDENT query window, the query is represented by the query collection in the following manner:

```
-  QUERYCOLLECTION(#STUDENT) =
     Array( Array( #STUDENT "       ") Array( #* "    ")
     Array( #SNAME ".P" ) Array( #AGE "         ")
     Array( #GPA ">= 3.5") Array( #MAJOR "       ");
```

The above returned example Array could be obtained at the ACTOR level if 'QUERYCOLLECTION(#STUDENT); <CR>' was typed into the ACTOR display window. The system is asked to show the representation of the STUDENT query array contained in the indexed collection that holds all query collections. The information is taken as symbols for the first element of each array and strings of characters for the second element instructions. A blank string is used to indicate no instructions on the object or attribute. The above example is pedagogical in nature, and it is not intended that the user would have access to the ACTOR system. The example merely shows the physical representation of the query object.

E. THE PHYSICAL DESCRIPTION

The data structures that support the GLAD interfaces have been discussed in great detail. The Array, Dictionary and Set classes, direct descendants of the Collection class, are ideally suited for supporting the GLAD interfaces. Essentially, the entire GLAD database application can be implemented as a Collection of Collections. The pedagogical University database example illustrated the use of specific ACTOR classes as data structures for supporting instances of GLAD objects. [Ref. 1:pp. 1-10]

Both physical and logical views of the GLAD data structures have been discussed. Data structures to support the window interfaces have been described in detail. Information contained in these data structures must be translated into extended SQL and ultimately primitive SQL to obtain data from the relational backend. Ultimately, data from the supporting data structures must be accessed in object oriented format, translated to relational format, and retranslated to object oriented format. The translation scheme and algorithm for the interfaces will be discussed in the subsequent chapter.

VII. ALGORITHMS FOR THE TRANSLATION

The window interfaces and data structures that will support GLAD have been discussed and analyzed in great detail in the previous chapters. Essentially, all of the objects and windows support the GLAD system with object oriented interfaces to the user and to the relational database system. GLAD can be thought of as a bridge that data travels on to the relational side and back to the object oriented side. However, this bridge is invisible to the user. From the user's vantage point, he only sees object oriented data.

A. THE TRANSPARENT LINK TO THE SYSTEM

Implementors of databases that use higher-level interfaces spend a great deal of time and effort in designing the database system in a manner that will allow the user to be ignorant of the design and implementation details. Therefore, when the user formulates a query, he does not have to know how the system is actually retrieving the data. With GLAD, efficient implementation of the higher level abstractions, supporting data structures and window interfaces is critical for obtaining an efficient database system with reasonable response time. Since the user is removed an additional level from the physical database, an efficient implementation of the interface is extremely important for retrieving data in a reasonable amount of time.

B. GLAD ON TOP OF SQL

As mentioned earlier, GLAD is to be built on top of a SQL based relational database system. It is anticipated that the SQL based system will function in the same manner as it would if it were the end user system with the SQL query language as the interface to the user. Instead of being directly supplied information by the user, the SQL based system will obtain information from GLAD and return data to GLAD in a manner that is entirely transparent to the user.

Consider a modular view of GLAD sitting on top of a SQL based system, DB2 by Date [Ref. 14:p. 103], and main components. A diagram of the system is shown in Figure 33.

A user of a SQL-based relational database system would perceive the database as base tables. In Date [Ref. 6:p. 103], the following definition of tables and views are given:

> A base table is a "real" table--i.e., a table that physically exists, in the sense that there exist physically stored records, and possibly physical indexes....By contrast, a view is a "virtual" table--i.e., a table that does not directly exist in storage.

With GLAD the user will be entirely shielded from the lower half of the diagram that uses SQL as the end user interface and tables and views to represent data entities. Instead of tuples in tables, the user perceives data as

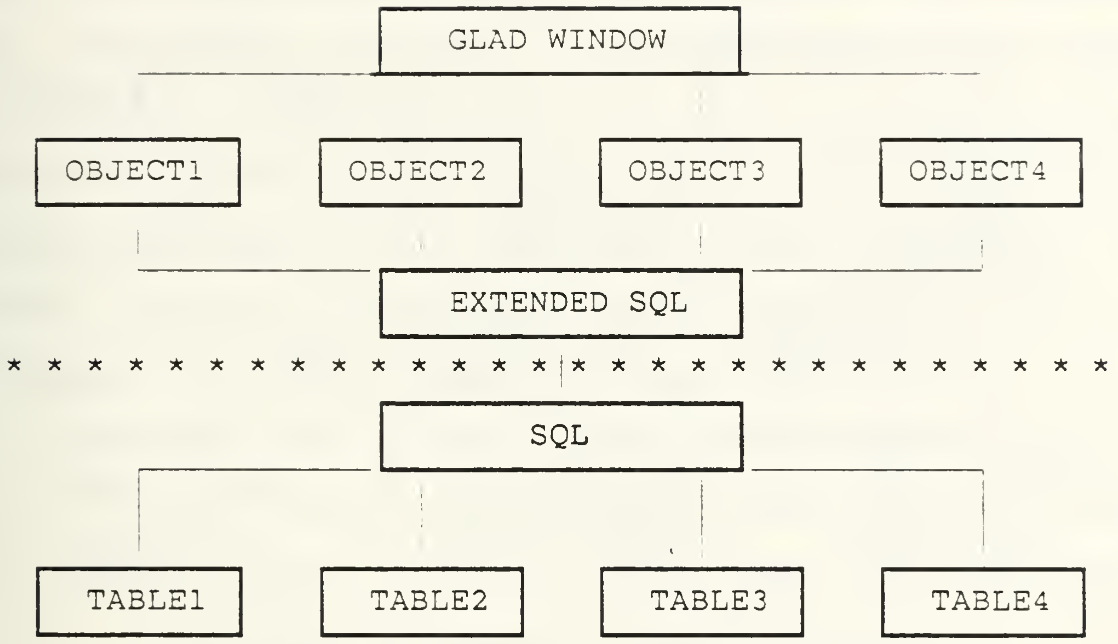


Figure 34. GLAD and SQL System Modules

graphics objects that maintain all the relevant facts and information about the data entities. Therefore, the user does not have to navigate through the rudimentary constructs of the relational SQL query language.

C. THE TRANSLATION SCHEME

A transparent link to the user can be formed with an effective translation scheme and efficient algorithms to the support the scheme. Consider the translation scheme of GLAD in Wu [Ref. 16:p. 5]. It is shown in Figure 35.

The left side of the diagram contains the data manipulation language scheme. The user formulates a database query with the query window. The data from the query is collected from the query window in an object query collection. Next,

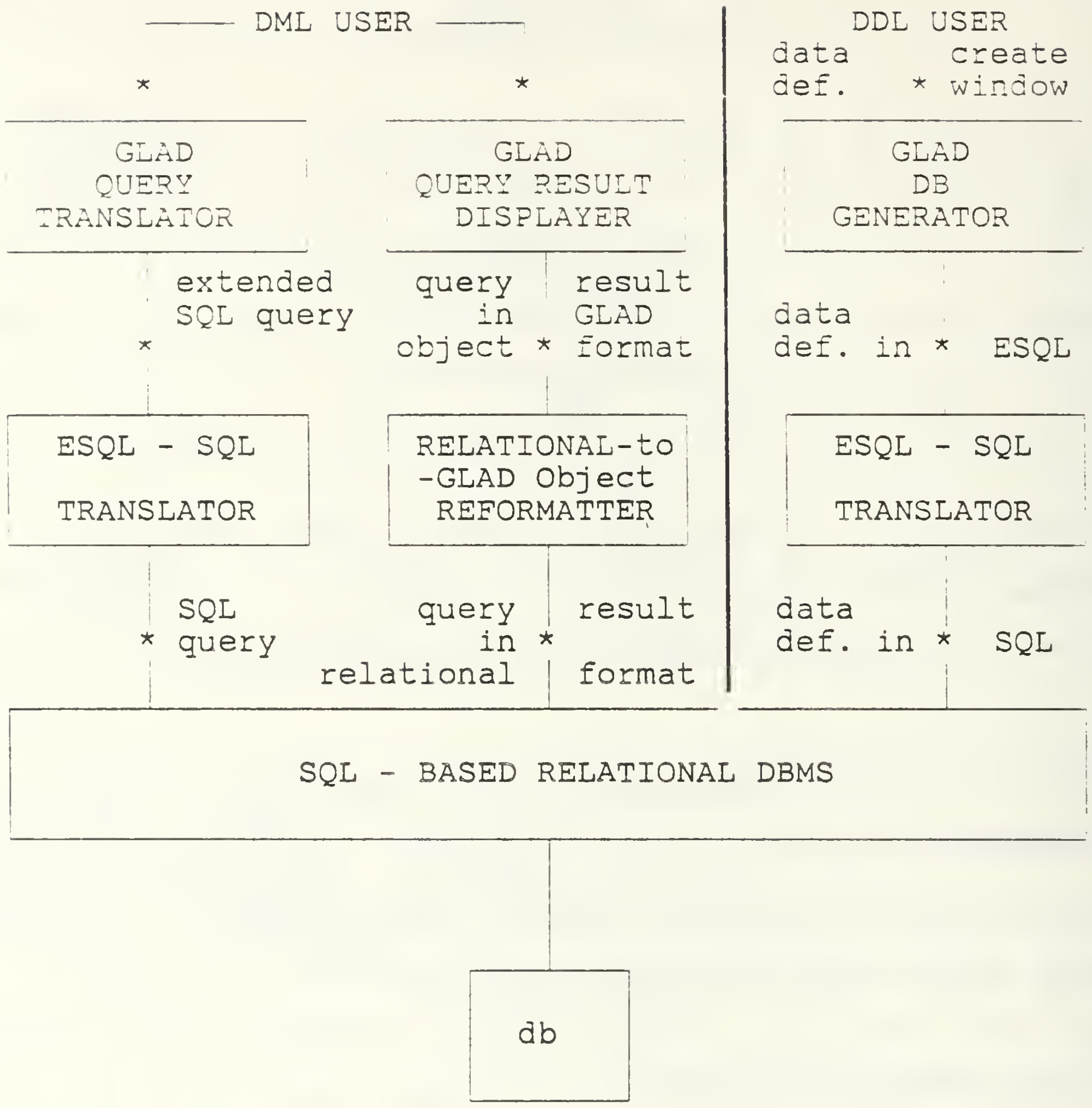


Figure 35. Translation Scheme

the query collection is translated into extended SQL syntax by the GLAD query translator. Then, the extended SQL query is finally translated into SQL syntax. The formatted SQL query is used by the relational database model to retrieve or select information stored in the database. The relational query result is sent to a relational-to-GLAD object reformatter to change relational query results to object oriented format. The query results, now in GLAD object format, are sent to the GLAD query result displayer. The result displayer takes the query result information and displays the results to the user in a result window.

The right side of the diagram contains the data definition language scheme. The user defines his database through the create database window. Information from the create window is collected by a create data base object and sent to the GLAD database generator. Data Definition is completed and the data is put into extended SQL syntax. The ESQL-SQL translator translates the extended SQL query and formulates the syntax for the SQL create table operation. The tables are then created and stored in the database. This diagram presents a high level view and summary of GLAD used in conjunction with a relational data base model.

D. AN ALGORITHM FOR THE TRANSLATION OF GLAD

As mentioned earlier, GLAD objects move across a bridge like interface to the relational system. By the time these objects arrive at the relational system they must be translated or converted to relational format. The relational system will only understand commands and queries in its native query language. Therefore, a generalized translation algorithm to accomplish conversion from object oriented format to relational format must be developed and implemented. Consider the translation algorithm contained below in Figure 36.

```
if (MORE_THAN_ONE_OBJECT_USED_IN_QUERY) then
    JOIN_OBJECTS;

for i := 1 to NO_PARTIAL_QUERIES_IN_QUERY do begin

    if (INSTRUCTIONS_ON_OBJECT_NAME(i)) then
    OBJ_OPERATIONS(i);

    while (EXECUTE_AND_DECODE(i)) do
    begin
             if (ASTERISK_FOR_ATTRIBUTE_NAME(i)) and
                  EXISTENTIAL_QUANTIFIER(i) then
        SPECIAL_OPS(i)
        else if (DOT_P_ON_ATTRIBUTE(i)) then
        RETRIEVE_SELECT(i)
        else if (NO_INSTRUCTIONS(i)) then NO_OP(i)
        else if (INSTRUCTIONS_ON_ATTRIBUTE(i)) then
            RELATIONAL_OPERATORS;
    end;

    if (not EXECUTE_AND_DECODE(i)) then
       RESET_GLOBAL_COUNTER;

end; { for i := 1 to NO_PARTIAL_QUERIES_IN_QUERY }
```

Figure 36. The GLAD Translation Algorithm

The algorithm of Figure 36 shall be used to make the translation to extended-relational format by accomplishing the correlation of specific object oriented constructs to relational constructs, and substituting or converting these constructs to extended SQL format.

E. AN ANALYSIS OF THE TRANSLATION ALGORITHM

The algorithm to translate the GLAD query divides the query collection into number of objects, object instructions and attribute instructions to formulate the SQL query on a line by line basis. The SQL translation is handled in a line by line manner to form the SELECT, FROM and WHERE lines that define the major components of a SQL query. Consider the translation of any GLAD query that consists of a query collection that has sub-queries on objects. The following major query items must be translated:

1. The number of objects in the query.
2. Any instructions on the object name.
3. Specific instructions for each attribute.

1. <u>If There are Multiple Query Objects</u>

If there are multiple objects used in the GLAD query, these objects must be joined as tables in the SQL query. Simple Objects can be directly correlated to a table that exists in the relational database. Complex Objects can be directly correlated to one primary table and one or many supporting identifier tables. It is anticipated that the create database function of GLAD will create primary relational tables that have the same name as the corresponding database object. Therefore, the translation will be able to substitute the object names directly into the FROM line of the SQL query. This process will be accomplished by the implementation of the following portion of the translation algorithm:

```
- if (MORE_THAN_ONE_OBJECT_USED_IN_QUERY) then
      JOIN_OBJECTS;
```

In a conventional manner, MORE_THAN_ONE_OBJECT_USED-_IN_ QUERY would be implemented as a function that returns a boolean value of true if multiple objects are used for the query. The query collection would be scanned by the function. If true was returned, a procedure called JOIN_OBJECTS would be utilized to formulate a relational join on the from line of the SQL query.

2. Each Object as a Partial Query

Each object that is used to formulate the GLAD query shall be considered to be a partial query for the GLAD query. Therefore, each of these objects must be decoded by the translator. The following portion of the algorithm steps through each of the objects in the algorithm:

```
- for i := 1 to NO_PARTIAL_QUERIES_IN_QUERY do
```

3. Decoding the Individual Objects

Inside of each iterative step of the above loop, the instructions on the objects and attribute names are decoded.

a. Decoding Object Instructions

First, consider decoding instructions on the object name. Instructions on attribute names are used to print out string messages in the query. These messages make the returned results more understandable. The following portion of the algorithm shall accomplish the translation of object instructions:

```
- if (INSTRUCTIONS_ON_OBJECT_NAME(i)) then
     OBJ_OPERATIONS(i);
```

If there are any instructions to be carried out on the object, such as a string to be returned with the retrieved values, they will be executed through a procedure named OBJ_OPERATIONS(i). This procedure will be called if a function called INSTRUCTIONS_ON_OBJECT_NAME(i) returns a boolean value of true. The function will evaluate to true if the attribute entry that holds object instructions does not contain a blank string.

b. Decoding Attribute Instructions

Each query collection has an array of ordered pairs that contain the attribute name and specific instructions that are to be performed on that attribute. The following portion of the algorithm steps through each attribute in the array and decodes the attribute instructions:

```
- while (EXECUTE_AND_DECODE(i)) do
```

This system shall continue to execute this portion of the algorithm until all of the attributes have been decoded. Essentially, EXECUTE_AND_DECODE(i) is a function that returns a boolean value of true as long as attribute instructions remain to be decoded. The attribute's are referenced in specific decode procedures by a global counter that is reset when each sub-query is decoded. After an attribute's object oriented instructions are decoded, control of the translation returns to the top of the while loop. While decoding attribute instructions, the translator keys on the following items for the translation:

1. An asterisk for the attribute name used in conjunction with an existential quantifier
2. A '.P' operation for an attribute instruction to indicate an attribute retrieval
3. A blank character string for attribute instructions
4. Relational operators for attribute instructions.

(1) <u>Asterisk and Existential Quantifier</u>. Special operations will be executed if both functions evaluate true in the following portion of the translation algorithm:

```
- if (ASTERISK_FOR_ATTRIBUTE_NAME(i)) and
     EXISTENTIAL_QUANTIFIER(i) then
     SPECIAL_OPS(i)
```

The SPECIAL_OPS(i) procedure shall formulate a nested select query from the pseudo attribute asterisk '*' containing instructions of '.EXIST'. However, if some type of instruction other than the existential quantifier is used with the '*' attribute, a nested select query will not be formed.

(2) <u>'.P' Operations for Retrieval</u>. If the attribute instruction field contains a '.P', then that particular attribute will be designated for retrieval as

indicated by the below listed portion of the translation algorithm:

```
- if (DOT_P_ON_ATTRIBUTE(i)) then RETRIEVE_SELECT(i)
```

DOT_P_ON_ATTRIBUTE(i) is a boolean function that will call the RETRIEVE_SELECT procedure if the function evaluates true. The function shall evaluate true when '.P' is contained in the character string for attribute instructions. In addition, it must be emphasized that '.P' can be qualified with further operations that the translator will handle.

(3) <u>No Instructions on the Attribute</u>. If the instruction field contains a blank character string, then a simple no-operation procedure is executed and control is returned to the main decode loop after the attribute counter is incremented. The following portion of the algorithm will execute the no-operation procedure if the boolean function evaluates to true:

```
- if (NO_INSTRUCTIONS_ON_ATTRIBUTE(i)) then NO_OP(i)
```

(4 <u>Relational Operators { =, <>, <, > }</u>. If instructions exist on the attribute besides those that have already been covered, the instructions will be relational operators. If the attribute instruction field contains equal to (=), not equal or unequal (<>), less than (<) or greater than (>), the INSTRUCTIONS_ON_ATTRIBUTE(i) function will evaluate to true. The RELATIONAL_OPERATORS procedure will be invoked to handle the attribute instructions.

F. FROM EXTENDED SQL TO SQL SYNTAX

As previously discussed, the correlations between the object oriented to extended SQL syntax are nearly a perfect one-to-one correspondence. However, the format of the extended SQL syntax is not suitable for the final SQL query. This extended syntax shall translate the complex data types of the extended query to atomic data types. Through the use of a join on a surrogate identifier table, the complex data type will be represented in the relational database as atomic data. Consider the following algorithm that is contained in Figure 37 and will be used by the ESQL to SQL translator.

```
for i := 1 to NUM_LINES_IN_EXTENDED_QUERY do begin
    if ESQL_Line_CONTAINS_COMPLEX_DATA(i) then
    begin
        TRANSLATE_ATOMIC_DATA_CONSTRUCTS(i);
        COMPLEX := true;
    end;
    if COMPLEX then
        JOIN_IDENTIFIER_TABLES_WITH_OBJECT_TABLE
    else
        SQL_EQUALS_ESQL
end;
```

Figure 37. Algorithm for ESQL to SQL

Essentially, the ESQL to SQL translator will be required to iterate through an array type structure that holds the complex data, parse the data, and substitute atomic constructs (i.e., DID: integer <-- MAJOR: DEPT). Then the surrogate identifier table will be substituted in the FROM line (i.e., FROM DEPT, IDDEPT <-- FROM DEPT).

G. A TRANSLATION EXAMPLE

Consider a simple example to illustrate the entire translation process. The database user wishes to make a STUDENT query that will retrieve all student names for students that have a grade point average of greater than or equal to 3.5 and are math majors. The first step in the query process is putting this plain English query information into the query window. Figure 38 illustrates the STUDENT query window after the user has entered the data.

STUDENT QUERY	
STUDENT	
*	
SNAME	.P
AGE	
GPA	>= 3.5
MAJOR	= 'MATH'

Figure 38. GLAD Query Window

After the user has entered the above information, a method that loads the query collection shall be utilized. The query collection shall contain only one element:

```
  -  QUERYCOLLECTION(#STUDENT) =
        Array( Array( #STUDENT "       ") Array( #* "    ")
        Array( #SNAME ".P" ) Array( #AGE "         ")
         Array( #GPA ">= 3.5") Array( #MAJOR "MATH");
```

Next, the STUDENT element will be translated by the GLAD query translator into ESQL syntax.

1. <u>Number of Objects in the Query</u>

The first step in the translation is determining if there is more than one object used in the query. In the example, there is only one object in the query. The function MORE_THAN_ONE_OBJECT_USED_IN_QUERY evaluates to false, and the procedure JOIN_OBJECTS is not called. Moreover, the ESQL FROM line will only have one table. At this point in the query the ESQL query array only has two elements that are not blank. Figure 39 contains the ESQL query after the FROM line has been translated.

ESQL ARRAY	
1	SELECT
2	FROM STUDENT
3	@@@@@@@@@@
4	@@@@@@@@@@
5	@@@@@@@@@@

Figure 39. ESQL Array during Query Translation

At this point, the attributes have not been decoded so the only lines that are resident in the ESQL array are the SELECT and FROM STUDENT lines. The other lines have been initialized to '@@@@@@@@@@' which is a sentinel for decoding.

2. Number of Partial Queries in Query

Each of the objects in the query collection represent a partial query. The loop with i := 1 to NO_PARTIAL_QUERIES_IN_QUERY will execute only once for the example query of Figure 38 because there is only one query object in the query collection.

3. Decoding the Object and Attribute Instructions

The loop while EXECUTE_AND_DECODE(i) will continue to execute until the function EXECUTE_AND_DECODE(i) evaluates to false.

a. Decoding the Object Instuctions

The query collection is checked to determine if there are object instructions that must translated. The first element in the query array is evaluated by the function INSTRUCTIONS_ON_OBJECT_NAME(i). The function will check to see if the instruction field has instructions. If the field is blank, the function will return a boolean value of true and call OBJ_OPERATIONS(i). In the example contained in Figure 38, there are no instructions on the attribute name. The function will evaluate to false.

b. Decoding the Attribute Instructions

Next, each of the attributes will be checked to determine if there are instructions for decoding. The while EXECUTE_AND_DECODE(i) loop will continue to execute until all the attribute instructions have been decoded. The * and AGE attributes contained in the query have no instructions. These attributes cause the NO_INSTRUCTIONS(i) function to evaluate true and call the NO_OP(i) procedure. The NO_OP(i) procedure increments the global counter but does not write to the ESQL query array. The second attribute, SNAME, causes the DOT_P_ON_OPERATION_ function to evaluate true. The translator analyzes the instruction field of SNAME and determines a '.P' operation must be performed on SNAME. The RETRIEVE_SELECT(i) procedure is called, and SNAME is written to the select line. At this point, the first two lines of the query have been formed, and the system could retrieve all of the STUDENT names from the STUDENT table. However, a condition still needs to be specified and the translation is still incomplete. Figure 40 shows the partially completed ERSQL query.

The fourth attribute, GPA, is translated, and the INSTRUCTIONS_ON_ATTRIBUTE(i) function evaluates true and calls the RELATIONAL_OPERATORS procedure. This procedure translates the instruction field and determines that a legal operation, '>=' (greater or equal than), is to be performed on the integer 3.5. Since this operation represents the

ESQL ARRAY		
1	SELECT	SNAME
2	FROM	STUDENT
3	@@@@@@@@@@	
4	@@@@@@@@@@	
5	@@@@@@@@@@	

Figure 40. ESQL Array after Two Lines are Translated

first condition of the query, the attribute operation is written to the third line of the query that will be composed of 'WHERE GPA >= 3.5'. Figure 41 shows the translated ESQL. This partial query represents the first three lines of the ESQL translation. The final attribute to be decoded is MAJOR. The MAJOR attribute is a complex attribute of type DEPT. For the ESQL translation, the complex data type is represented like any other attribute in the ESQL query. The INSTRUCTIONS_ON_ATTRIBUTE(i) function is evaluated to true and calls the RELATIONAL_OPERATORS procedure. The RELATIONAL_OPERATORS procedure determines that a valid operation, '=' (equals), is to be performed on the MAJOR attribute. This operation will be translated to 'AND MAJOR = 'MATH' ' and written to the fourth line of the ESQL query array. Figure 42 displays the final ESQL translation after all attributes have been decoded.

ESQL ARRAY	
1	SELECT SNAME
2	FROM STUDENT
3	WHERE GPA >= 3.5
4	@@@@@@@@@@
5	@@@@@@@@@@

Figure 41. ESQL Query with Three Lines Translated

ESQL ARRAY	
1	SELECT SNAME
2	FROM STUDENT
3	WHERE GPA >= 3.5
4	AND MAJOR = 'MATH'
5	@@@@@@@@@@

Figure 42. ESQL Array after Translation is Complete

The ESQL query is ready to be translated to primitive SQL syntax for the final SQL query. The ESQL query shall be sent to the ESQL to SQL translator to accomplish the final translation.

4. ESQL TO SQL

For the final translation, the ESQL to SQL translator shall iterate through the query array and determine if a line contains a complex data type. The first four lines in the ESQL query can be substituted without change to final SQL query.

The last line of the query has a non-atomic attribute type and a complex translation must occur. The SQL translator determines that MAJOR is not atomic by consulting a table that holds all complex data types. MAJOR is of type DEPT. The ESQL_Line_CONTAINS_COMPLEX_DATA(i) function evaluates to true, and the procedure TRANSLATE_ATOMIC_DATA_-CONSTRUCTS(i) will translate the final line to read 'AND DIDNAME = 'MATH' '. DIDNAME is an identifer of type string that will be compatible with primitive SQL constructs.

The final step in the translation process is the procedure JOIN_IDENTIFIER_TABLES_WITH_OBJECT_TABLE adding the identifier table, IDTOWN, to the FROM line. IDTOWN acts as a surrogate table that is joined with the STUDENT table to indirectly represent complex data as atomic data. The completed SQL query is shown below in Figure 43. Now the SQL query can be delivered to the relational system to retrieve the desired student names.

The results of the query will be returned to the translator in set type format and retranslated to object-oriented format for user view.

SQL ARRAY		
1	SELECT	SNAME
2	FROM	STUDENT, IDDEPT
3	WHERE	GPA >= 3.5
4	AND	DIDNAME = 'MATH'
5	@@@@@@@@@@	

Figure 43. SQL Array after Translation is Complete

VIII. CONCLUSION

The need for a user friendly graphics interface to the relational database system has been validated by examining the deficiencies and limitations of the current relational database systems. The deficiencies and limitations of these systems are caused by the relational model's lack of semantic capability. Relational systems lack semantic power because they are based on poor semantic data models. Systems that support specific higher level abstraction concepts have the power, flexibility and capability to provide a user friendly environment and extended relational capabilities. These systems should support aggregation, association, generalization and classification.

Inadequate semantic capability is not the only reason that relational systems are not well suited for the entire population of database users. Moreover, standard relational query languages are untenable for naive and inexperienced database users. People who generally do not have formal computer education or training are often in administrative, clerical and technical positions that require the use of computers to do their jobs. In fact, computer technology has experienced wide spread proliferation, and computers have permeated our society. Therefore, it is important that non-computer professionals who need computer technology to perform every day job functions be given the best possible user interfaces for their working databases.

Research has determined that the previously discussed relational query languages are entirely unsuitable for these types of individuals. Tuple calculus and relational algebra based systems may provide and effective user interface with QUEL and SQL for the mathematician or computer scientist. The mathematical query concepts can be associated by the user to already familiar math or computer concepts. Unfortunately, the naive user will probably not be familiar with these concepts.

From a human factors design point of view, a system must provide the user with functional items that can be associated to familiar concepts. Database entities and subentities can be effectively associated with simple graphics objects. GLAD utilizes rectangles and lines to make these correlations between the real world entity and abstract object.

A. EXTENDING THE RELATIONAL SYSTEM

More than a decade of arduous research has validated the existing relational database technology. Relational theory has provided a sound theoretical basis for the current relational systems. It would be unwise to totally abandon this technology and start from ground zero to develop a new system. Perhaps, a major technological break through in computer architecture would merit a complete database system redesign. Furthermore, most of today's database technology is based on research that was done nearly 20 years ago at IBM research center in San Jose, California. In addition to Codd's relational model, considerable advances were made in all aspects of database technology that are entirely relevant to today's systems.

1. A Stable Database Technology

Today, computers have become relatively inexpensive and have been designed to be easier to learn and use. Hardware component prices have consistently plummeted in value. In the last decade, memory has become orders of magnitude cheaper. Despite advances in both hardware and software, database theory has fundamentally remained consistent since the advent of the relational model.

2. Extend Existing Database Technology

Therefore, until a major technological break through has been accomplished, such as the development of an inexpensive secondary storage device that is orders of magnitude faster than disks, the wise approaches for improving database systems are centered around extending the existing data base technology. GLAD extends the existing relational system through graphical user interfaces that transport data and results to and from the relational system.

B. THE OBJECT ORIENTED SYSTEM

When implemented, GLAD's power, flexibility and user friendliness will provide the best possible interface to the user. Much of this capability is derived from the object oriented approach. ACTOR, the object oriented language used for GLAD implementation, will provide the window interfaces to the user. The capabilities of GLAD's user interfaces will be derived from ACTOR's object-oriented classes and methods. ACTOR's object oriented classes will supply GLAD with the graphical objects that represent the database. In addition, GLAD shall utilize ACTOR's Collection descendants to define data structures that will effectively support the user interfaces.

The relationship between ACTOR and GLAD is analogous to the relationship between SQL and embedded PL1 or Cobol. However, a major difference is that GLAD can be represented in a much more natural manner than SQL embedded in PL1.

C. FUTURE GLAD APPLICATIONS

GLAD has many potential uses for future applications. It has potential military, administrative, educational and training possibilities. The easy to use and learn interfaces will make the system available to the widest range of database users.

1. Military Applications

GLAD has tremendous potential for a wide variety of applications. In particular, GLAD seems well suited to a wide variety of military uses and applications. Military office usage of database could be aided significantly by providing good graphics interfaces to military administrative and data processing clerks. Military personnel often report to their commands and must be immediately integrated into the work environment without any significant on-the-job training. In addition, there are high turnover rates in most units due to transfers, discharges and reenlistments. Personnel must constantly be retrained on systems. GLAD could certainly help a clerk, that did not have computer experience, to quickly become a proficient database user.

In a tactical environment, GLAD could be very valuable to the field commander. It could be used to formulate queries for portable PC based military data bases. Intelligence, logistic and historical information could be quickly accessed by troops and passed to their superiors for important tactical and strategic evaluation.

2. Database Educational and Training Requirements

GLAD could be useful as a tutorial for teaching naive users to use other database systems. Since GLAD is an extension of the relational database, a one-to-one correspondence can be established for the GLAD items that are substituted and translated to SQL items. This correspondence can be used to show the user, in piece meal fashion, how the GLAD query corresponds to the SQL query.

D. REMARKS ON IMPLEMENTATION

GLAD implementation of graphics interfaces is an on going project with considerable work remaining in striking the ACTOR code for the higher level interfaces. However, much work has already been accomplished. In particular, the higher level abstractions have been formulated. Conceptually, GLAD is entirely ready to be implemented.

1. Mastering ACTOR

The most significant challenge to the implementors may be mastering the ACTOR programming language. The object oriented approach requires considerable departure from previous ways of forming code and writing programs. Many programmers have developed their programming strategies in ways that will require significant adaptation to object oriented format. Although ACTOR programs will generally be much shorter than a corresponding conventional, higher level language programs, the ACTOR code may actually require more thought than conventional code. The learning curve on object oriented concepts seems to be high.

2. Interfaces and QueryCollections

However, once these concepts are mastered, object oriented programming power can provide methods for graphical interfaces that can not be obtained with conventional languages. ACTOR maintains graphical methods, in a compact and precise manner, that are readily available with object oriented languages. ACTOR will be ideal for developing methods and classes that will define the data structures that will support the ACTOR interfaces. The interfaces shall be a collection point for data, and QueryCollection's will serve as the transportation medium of this data to and from the user.

3. The Bus Station Analogy

The bus station analogy should prove to be useful in illustrating the interface implementations. A bus station can be thought of as a collecting point for people that need to ride the bus to a given destination. The query window is a collecting point for data that must ultimately be transferred to the relational system and back to the user. The bus is the vehicle of transportation for taking the people to their destination. In fact, there are two types of buses:

1. extended buses
2. native buses.

The extended buses take the people long distances from state to state and city to city. The native or local buses are driven by residents of the local community and take the bus riders, who have completed their extended bus ride, to the ultimate inner city or community destination. In other words, the native buses are short range buses, and the extended buses are long range buses.

The methods that translate the GLAD query information to extended SQL information can be compared to the extended bus that takes the user to the city of his intermediate destination. The methods that translate the extended SQL to native relational SQL can be compared to the native or local bus that takes the user to his ultimate destination.

E. EPILOGUE

The system formed from GLAD and a relational database will provide a powerful and easy to use interface to the widest possible audience of database users. GLAD represents the natural evolution of the relational system. In essence, the GLAD approach to database is evolutionary vice revolutionary. GLAD is not revolutionary because it does not propose a new database model with a new architecture to support the implementation. It is evolutionary because it takes a model that has already been validated and builds the interface on top of that model. In fact, GLAD could be implemented on the network, hierarchical and inverted list data models. Since GLAD is not based on any specific kind of model, it has a great deal of flexibility for serving as a bridge between the user and database. This approach has not been entirely explored and many systems based on this concept should be implemented before graphics extensions have been truly maximized as an effective man to machine interface. Until technology surpasses the capabilities of existing computer systems, the evolutionary approach to designing new database systems will provide the highest yield for the least investment.

APPENDIX A

A SIMULATED GLAD TRANSLATOR

```
program ObjectTranalator;

{ This program illustrates the use of the algorithm that
  will be used to translate object query windows to SQL
  syntax. The program is written in Borland Turbo
  Pascal 3.0 and executed on IBM compatible PC/XT micro-
  computer.
  FOR THESIS RESEARCH IN CONJUNCTION WITH GLAD
  Author Paul D. Grenseman, NPS/CS-71 - March 1988 }

type

    OPERATION = string[16];
       { Used to hold relational operators for attributes
         or sub-objects that are queried }

    NAME = string[30];
       { Used to hold the NAME of the objects }

    OPERATIONS = array[1..10] of operation;
       { attribute operations of the query }

    NAMES = array[1..10] of NAME;
       { The NAMES of the objects that will be queried }

    INSTRUCTIONS = string[16];
       { INSTRUCTIONS to be performed on the entire
         object such as select all attributes or count
         all object attributes }

    ABOUT_OBJECTS = record
        OBJECT_NAME: NAME;
        OBJECT_INST: INSTRUCTIONS;
    end;

    ABOUT_ATTRIBUTES = record
        ATTR_NAME: NAMES;
        ATTR_OPERATION: OPERATIONS;
    end;



    PARTIAL_QUERY = record
        OBJECT_INFO: ABOUT_OBJECTS;
        ATTRIBUTES: ABOUT_ATTRIBUTES;
        IS_QUERIED: boolean;
    end;

       { A query that is not joined or completed. The
         user can elect to make a partial query complete
         by selecting the QUIT option }

    OBJECTS = record
        NAME: string[20];
        ATTRIBUTES: array[1..10] of string[20];
    end;
        { The Pascal simulation of the ACTOR object.
          < Object-declaration > ::=
           DEFOBJ < object_name >
              < attribute-declaration >+
           ENDOBJ }

    OBJECT_QUERY_WINDOW = array[1..10] of PARTIAL_QUERY;
    { Partial Queries to be joined or nested }

    OBJ_STR = string[40];

    JOBS = (FACULTIES, SECRETARIES);
    { Possible Employee Jobs }

    DEPT_SET = (MATH, COMPUTER, MUSIC, PHYSICS, HISTORY);
    { Departements the students can major in }

    QUERY_STR = string[50];
    { One line of the TRANSLATED object query }

    SQL_QUERY = array[1..40] of QUERY_STR;
    { The translated sequel query in relation syntax }

    EXTENDED_SQL_QUERY = array[1..40] of QUERY_STR;
    { The translated object query in extended
      SQL syntax }


const
    SENTINEL = '@@@@@@@@@@';
    { used during decoding process }

var
    ITH_ATTRIBUTE: integer;
    { The Ith attribute in the query it is used to
      determine the placement of WHERE & AND }

    e,
    X: integer;
    { Subscript for array of attributes }
```

```
NO_SELECTS: integer;
{ Number of nested selects in partial query }

i: integer;
{ Loop control variable for query }

JOIN: boolean;
{ Global boolean variable to indicate that
  objects in a query are joined }

F: char;
{ choose F or f to exit query process }
FINISHED,
{ FINISH set by f or F }
MORE_THAN: boolean;
{ more than one sub-query }


MAJOR, JOBTYPE, DEPT, MEMBERS, WORKSFOR: boolean;
{ Used for decoding SET TYPES }

NO_PARTIAL_QUERIES_IN_QUERY: integer;

STUDENT_QUERY, EMPLOYEE_QUERY, COMMITTEE_QUERY,
FACULTY_QUERY,
{ Simulated Object Query display windows }

PARTIAL_QUERY_OBJECTS: PARTIAL_QUERY;

OBJECT_NAME_STRING: OBJ_STR;

FACULTY, STUDENT, EMPLOYEE, COMMITTEE: OBJECTS;
{ Simulated GLAD University DB Objects }

QUERY: OBJECT_QUERY_WINDOW;
{ The Completed Object Query to be translated}

ERSQL: EXTENDED_SQL_QUERY;
{ The Sequel Query with COMPLEX TYPES }

SQL: SQL_QUERY;
{ The Translated Sequel Query in Relational format }

FSQL: text;
{ The text file that holds object, extended and
  relational queries }
```

```
procedure INITIALIZE_GLOBALS;
begin     { INITIALIZE_GLOBALS }
{ Set all Gobal variables to initial values }
    X := 1;
    e := 1;  { All arrays subscript start at one }
    JOIN := false;
    MORE_THAN := false;
    DEPT := false; MAJOR := false; WORKSFOR := false;
    JOBTYPE := false; MEMBERS := false;
    NO_PARTIAL_QUERIES_IN_QUERY := 0;
    STUDENT.NAME := '          ';
    COMMITTEE.NAME := '          ';
    FACULTY.NAME := '          ';
    EMPLOYEE.NAME := '          ';

    for i := 1 to 10 do begin
        QUERY[i].IS_QUERIED := false;
        STUDENT.ATTRIBUTES[i] := '          ';
        COMMITTEE.ATTRIBUTES[i] := '          ';
        FACULTY.ATTRIBUTES[i] := '          ';
        EMPLOYEE.ATTRIBUTES[i] := '          ';
    end;   { for i := 1 to 10 do }

    for i := 1 to 40 do begin
        ERSQL[i] := SENTINEL;
        SQL[i] := SENTINEL;
    end;   { for i := 1 to 40 do }

end;     { INITIALIZE_GLOBALS }


procedure LOAD_OBJECTS;
{ Simulated Objects loaded with attributes/sub-objects }

    procedure LOAD_STUDENT;
    begin { LOAD_STUDENT }
        STUDENT.NAME := 'STUDENT';
        STUDENT.ATTRIBUTES[1] := '*';
        STUDENT.ATTRIBUTES[2] := 'SNAME';
        STUDENT.ATTRIBUTES[3] := 'ADDRESS';
        STUDENT.ATTRIBUTES[4] := 'SSNO';
        STUDENT.ATTRIBUTES[5] := 'GPA';
        STUDENT.ATTRIBUTES[6] := 'MAJOR';
    end; { LOAD_STUDENT }

    procedure LOAD_COMMITTEE;
    begin { LOAD_COMMITTEE }
        COMMITTEE.NAME := 'COMMITTEE';
        COMMITTEE.ATTRIBUTES[1] := 'CNAME';
        COMMITTEE.ATTRIBUTES[2] := 'MEMBERS';
        COMMITTEE.ATTRIBUTES[3] := 'PURPOSE';
    end; { LOAD_COMMITTEE }
    procedure LOAD_FACULTY;
    begin { LOAD_FACULTY }
        FACULTY.NAME := 'FACULTY';
        FACULTY.ATTRIBUTES[1] := 'FNAME';
        FACULTY.ATTRIBUTES[2] := 'AGE';
        FACULTY.ATTRIBUTES[3] := 'WorksFor';
    end; { LOAD_FACULTY }



    procedure LOAD_EMPLOYEE;
    begin { LOAD_EMPLOYEE }
        EMPLOYEE.NAME := 'EMPLOYEE';
        EMPLOYEE.ATTRIBUTES[1] := 'ENAME';
        EMPLOYEE.ATTRIBUTES[2] := 'PAY';
        EMPLOYEE.ATTRIBUTES[3] := 'DEPT';
        EMPLOYEE.ATTRIBUTES[4] := 'JobType';
    end; { LOAD_EMPLOYEE }

begin     { LOAD_OBJECTS }

LOAD_STUDENT;
LOAD_COMMITTEE;
LOAD_FACULTY;
LOAD_EMPLOYEE;

end;    { LOAD_OBJECTS }


procedure LOAD_WINDOWS_WITH_OBJECTS;
{ Query windows loaded with object information }

   procedure LOAD_STUDENT_QUERY;
   var
       W: integer;
   begin { LOAD_STUDENT_QUERY }
       STUDENT_QUERY.OBJECT_INFO.OBJECT_NAME :=
            STUDENT.NAME;
       STUDENT_QUERY.OBJECT_INFO.OBJECT_INST :=
            '         ';
       STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[1] :=
            STUDENT.ATTRIBUTES[1];
       STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[2] :=
            STUDENT.ATTRIBUTES[2];
       STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[3] :=
            STUDENT.ATTRIBUTES[3];
       STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[4] :=
            STUDENT.ATTRIBUTES[4];
       STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[5] :=
              STUDENT.ATTRIBUTES[5];
    STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[6] :=
           STUDENT.ATTRIBUTES[6];
    STUDENT_QUERY.IS_QUERIED := false;

    for W := 1 to 6 do
    STUDENT_QUERY.ATTRIBUTES.ATTR_OPERATION[W] :=
             '         ';
end; { LOAD_STUDENT_QUERY }


procedure LOAD_FACULTY_QUERY;
var
   W: integer;
begin { LOAD_FACULTY_QUERY }
    FACULTY_QUERY.OBJECT_INFO.OBJECT_NAME :=
          FACULTY.NAME;
    FACULTY_QUERY.OBJECT_INFO.OBJECT_INST :=
          '          ';
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[1] :=
          FACULTY.ATTRIBUTES[1];
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[2] :=
          FACULTY.ATTRIBUTES[2];
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[3] :=
          FACULTY.ATTRIBUTES[3];
    FACULTY_QUERY.IS_QUERIED := false;
    for W := 1 to 3 do
    FACULTY_QUERY.ATTRIBUTES.ATTR_OPERATION[W] :=
          '          ';
 end; { LOAD_FACULTY_QUERY }


procedure LOAD_COMMITTEE_QUERY;
var
   W: integer;
begin { LOAD_COMMITTEE_QUERY }
    COMMITTEE_QUERY.OBJECT_INFO.OBJECT_NAME :=
         COMMITTEE.NAME;
    COMMITTEE_QUERY.OBJECT_INFO.OBJECT_INST :=
         '          ';
    COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[1] :=
         COMMITTEE.ATTRIBUTES[1];
    COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[2] :=
         COMMITTEE.ATTRIBUTES[2];
    COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[3] :=
         COMMITTEE.ATTRIBUTES[3];
    COMMITTEE_QUERY.IS_QUERIED := false;
    for W := 1 to 3 do
    COMMITTEE_QUERY.ATTRIBUTES.ATTR_OPERATION[W] :=
         '         ';
 end; { LOAD_COMMITTEE_QUERY }
```

```
    procedure LOAD_EMPLOYEE_QUERY;
    var
       W: integer;
    begin { LOAD_EMPLOYEE_QUERY }
        EMPLOYEE_QUERY.OBJECT_INFO.OBJECT_NAME :=
             EMPLOYEE.NAME;
        EMPLOYEE_QUERY.OBJECT_INFO.OBJECT_INST :=
             '          ';
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[1] :=
             EMPLOYEE.ATTRIBUTES[1];
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[2] :=
             EMPLOYEE.ATTRIBUTES[2];
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[3] :=
             EMPLOYEE.ATTRIBUTES[3];
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[4] :=
             EMPLOYEE.ATTRIBUTES[4];
        EMPLOYEE_QUERY.IS_QUERIED := false;
        for W := 1 to 4 do
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_OPERATION[W] :=
             '         ';
    end; { LOAD_EMPLOYEE_QUERY }

begin { LOAD_WINDOWS_WITH_OBJECTS }

    LOAD_STUDENT_QUERY;
    LOAD_EMPLOYEE_QUERY;
    LOAD_FACULTY_QUERY;
    LOAD_COMMITTEE_QUERY;

end; { LOAD_WINDOWS_WITH_OBJECTS }



procedure DISPLAY_QUERY_OPTIONS;
{ Screen Display that allows you to exit the query
  process or choose various query windows to be
  selected, nested, or joined }

var
    SUBSCRIPT, COUNT, V: integer;
    Q, OPTION: char;
    P: PARTIAL_QUERY;
    S_INST, C_INST, E_INST, F_INST: boolean;
```

```
procedure CLEAR (INST:boolean);
 { Just blank Screen of old information }
 begin { CLEAR }
     if INST then begin
         writeln; writeln;
         write ('REVIEW YOUR QUERY AND HIT ENTER':50);
         readln (Q);
         clrscr;
     end; { INST }
     writeln;
 end;  { CLEAR }


 procedure DISPLAY_OBJECT_INSTRUCTIONS( INST:BOOLEAN;
                                   D:PARTIAL_QUERY );
 begin { DISPLAY_OBJECT_INSTRUCTIONS }
 { You may select special instructions '*' to select
   all attributes or 'COUNT(*)' to count all rows of
   object that meets specified conditions }

     if not INST then
         writeln('*':24 )
     else
         begin
              writeln (D.OBJECT_INFO.OBJECT_INST:22,
              ' *');
              writeln (FSQL,
              D.OBJECT_INFO.OBJECT_INST:22, ' *');
          end;
 end;  { DISPLAY_OBJECT_INSTRUCTIONS }


 procedure DISPLAY_ATTRIBUTE_OPERATIONS( INST:BOOLEAN;
                                    D:PARTIAL_QUERY );
 begin { DISPLAY_ATTRIBUTE_OPERATIONS }
 { Operations on attributes such as >, <>, =, >=, <=
   will be displayed to user after they are typed in }

     COUNT := COUNT + 1;
     if not INST then
         writeln('*':24 )
     else
         begin
          writeln
              (D.ATTRIBUTES.ATTR_OPERATION[COUNT]:16,
              '*':8);
              writeln (FSQL,
              D.ATTRIBUTES.ATTR_OPERATION[COUNT]:16,
              '*':8);
         end;
  end;  { DISPLAY_ATTRIBUTE_OPERATIONS }
```

```
{ The blank Query window is initially displayed
   to the user. He is asked to input instructions
   for objects and relational operations for
   attributes. To skip Object Instructions or an
   Attribute, hit enter. }

procedure DISPLAY_STUDENT;
begin { DISPLAY_STUDENT }
    COUNT := 0;
    if S_INST then
      begin
        STUDENT_QUERY := P;
        writeln(FSQL,
        '*************************
         **************************':60);
         write (FSQL,'*':10,
         STUDENT_QUERY.OBJECT_INFO.OBJECT_NAME:25);
         writeln(FSQL,' QUERY ', '*':18);
         writeln(FSQL,
         '*************************
          **************************':60);
        end;
        writeln('*************************
                 **************************':60);
        write('*':10,
        STUDENT_QUERY.OBJECT_INFO.OBJECT_NAME:25);
        writeln(' QUERY ', '*':18);
        writeln('*************************
                 **************************':60);
        if S_INST then
         write (FSQL,
         '*':10,
         STUDENT_QUERY.OBJECT_INFO.OBJECT_NAME:16,
         '*':10);

         write ('*':10,
         STUDENT_QUERY.OBJECT_INFO.OBJECT_NAME:16,
         '*':10);

         DISPLAY_OBJECT_INSTRUCTIONS( S_INST,
         STUDENT_QUERY );

         if S_INST then
         write (FSQL,
         '*':10,
         STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[1]:6,
         '*':20);
```

```
 write ('*':10,
 STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[1]:6,
 '*':20);
 DISPLAY_ATTRIBUTE_OPERATIONS( S_INST,
 STUDENT_QUERY );

 if S_INST then
 write (FSQL,
 '*':10,
 STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[2]:7,
 '*':19);

 write ('*':10,
 STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[2]:7,
 '*':19);
 DISPLAY_ATTRIBUTE_OPERATIONS( S_INST,
 STUDENT_QUERY );

 if S_INST then
 write (FSQL,
 '*':10,
 STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[3]:9,
 '*':17);

 write ('*':10,
 STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[3]:9,
 '*':17);

 DISPLAY_ATTRIBUTE_OPERATIONS( S_INST,
 STUDENT_QUERY );

if S_INST then
write (FSQL,
'*':10,
STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[4]:6,
'*':20);

write ('*':10,
STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[4]:6,
'*':20);

DISPLAY_ATTRIBUTE_OPERATIONS( S_INST,
STUDENT_QUERY );

if S_INST then
write (FSQL,
'*':10,
STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[5]:5,
'*':21);
```

```
        write ('*':10,
        STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[5]:5,
        '*':21);

        DISPLAY_ATTRIBUTE_OPERATIONS( S_INST,
        STUDENT_QUERY );

        if S_INST then
        write (FSQL,
        '*':10,
        STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[6]:7,
        '*':19);
        write ('*':10,
        STUDENT_QUERY.ATTRIBUTES.ATTR_NAME[6]:7,
        '*':19);
        DISPLAY_ATTRIBUTE_OPERATIONS( S_INST,
                                     STUDENT_QUERY );
        if S_INST then
        begin
        writeln(FSQL,
        '*************************
         *************************':60);
        writeln(FSQL)
        end;
               writeln('*************************
                 *************************':60);

        if S_INST then begin
            SUBSCRIPT := SUBSCRIPT + 1;
            QUERY[SUBSCRIPT] := STUDENT_QUERY;
            QUERY[SUBSCRIPT].IS_QUERIED := true;
        end;  { if S_INST }

    CLEAR (S_INST);
    P := STUDENT_QUERY;

end; { DISPLAY_STUDENT }

 procedure DISPLAY_EMPLOYEE;
 begin { DISPLAY_EMPLOYEE }
    COUNT := 0;
    if E_INST then
        begin
            EMPLOYEE_QUERY := P;
            writeln( FSQL,
            '*************************
             *************************':60);
             write  ( FSQL,
            '*':10,
            EMPLOYEE_QUERY.OBJECT_INFO.OBJECT_NAME:25);
            writeln( FSQL,
            ' QUERY ','*':18);
        writeln( FSQL,
        '**************************
         **************************':60);
    end;
    writeln('*************************
             **************************':60);
    write  ('*':10,
    EMPLOYEE_QUERY.OBJECT_INFO.OBJECT_NAME:25);
    writeln(' QUERY ','*':18);
    writeln('*************************
             **************************':60);

    if E_INST then
    write (FSQL,
    '*':10,
    EMPLOYEE_QUERY.OBJECT_INFO.OBJECT_NAME:16,
    '*':10);

    write ('*':10,
    EMPLOYEE_QUERY.OBJECT_INFO.OBJECT_NAME:16,
    '*':10);

    DISPLAY_OBJECT_INSTRUCTIONS( E_INST,
    EMPLOYEE_QUERY );

    if E_INST then
        write (FSQL,
        '*':10,
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[1]:7,
        '*':19);

    write ('*':10,
    EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[1]:7,
    '*':19);

    DISPLAY_ATTRIBUTE_OPERATIONS( E_INST,
    EMPLOYEE_QUERY );

    if E_INST then
    write (FSQL,
    '*':10,
    EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[2]:5,
    '*':21);

    write ('*':10,
    EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[2]:5,
    '*':21);

    DISPLAY_ATTRIBUTE_OPERATIONS( E_INST,
                                  EMPLOYEE_QUERY );
        if E_INST then
        write (FSQL,
        '*':10,
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[3]:6,
        '*':20);

        write ('*':10,
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[3]:6,
        '*':20);
        DISPLAY_ATTRIBUTE_OPERATIONS( E_INST,
                                      EMPLOYEE_QUERY );
        if E_INST then
        write(FSQL,
        '*':10,
        EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[4]:9,
        '*':17);

    write ('*':10,
    EMPLOYEE_QUERY.ATTRIBUTES.ATTR_NAME[4]:9,
    '*':17);

    DISPLAY_ATTRIBUTE_OPERATIONS( E_INST,
                                  EMPLOYEE_QUERY );
    if E_INST then
    begin
        writeln(FSQL,
        '****************************
         *************************':60);
        writeln(FSQL);
    end;
           writeln('*************************
             **************************':60);

    if E_INST then begin
    SUBSCRIPT := SUBSCRIPT + 1;
    QUERY[SUBSCRIPT] := EMPLOYEE_QUERY;
    QUERY[SUBSCRIPT].IS_QUERIED := true;
    end;  { if E_INST }

    CLEAR (E_INST);
    P := EMPLOYEE_QUERY;

end; { DISPLAY_EMPLOYEE }


procedure DISPLAY_FACULTY;
begin { DISPLAY_FACULTY }
    COUNT := 0;
    if F_INST then
        begin
            FACULTY_QUERY := P;
            writeln(FSQL,
        '*************************
         ***************************':60);
        write  (FSQL, '*':10,
        FACULTY_QUERY.OBJECT_INFO.OBJECT_NAME:25);
        writeln(FSQL,
        ' QUERY ','*':18);
        writeln(FSQL,
        '***************************
         ************************':60);
    end;
    writeln('*************************
    **************************':60);
    write  ('*':10,
    FACULTY_QUERY.OBJECT_INFO.OBJECT_NAME:25);
    writeln(' QUERY ','*':18);
    writeln('************************
    ***************************':60);

if F_INST then
    write (FSQL,
    '*':10,
    FACULTY_QUERY.OBJECT_INFO.OBJECT_NAME:16,
    '*':10);
    write ('*':10,
    FACULTY_QUERY.OBJECT_INFO.OBJECT_NAME:16,
    '*':10);
    DISPLAY_OBJECT_INSTRUCTIONS( F_INST,
    FACULTY_QUERY );
if F_INST then
    write (FSQL,
    '*':10,
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[1]:7,
    '*':19);
    write ('*':10,
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[1]:7,
    '*':19);
    DISPLAY_ATTRIBUTE_OPERATIONS( F_INST,
    FACULTY_QUERY );
if F_INST then
    write (FSQL,
    '*':10,
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[2]:5,
    '*':21);
    write ('*':10,
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[2]:5,
    '*':21);
    DISPLAY_ATTRIBUTE_OPERATIONS( F_INST,
    FACULTY_QUERY );
if F_INST then
    write (FSQL,
    '*':10,
    FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[3]:10,
        '*':16);
        write ('*':10,
        FACULTY_QUERY.ATTRIBUTES.ATTR_NAME[3]:10,
        '*':16);
        DISPLAY_ATTRIBUTE_OPERATIONS( F_INST,
        FACULTY_QUERY );
    if F_INST then
    begin
        writeln(FSQL,
        '***************************
         ***************************':60);
        writeln(FSQL);
    end;
    writeln('**************************
             **************************':60);
    writeln;

     if F_INST then begin
        SUBSCRIPT := SUBSCRIPT + 1;
        QUERY[SUBSCRIPT] := FACULTY_QUERY;
        QUERY[SUBSCRIPT].IS_QUERIED := true;
    end;  { if F_INST }

    CLEAR(F_INST);
    P := FACULTY_QUERY;
end; { DISPLAY_FACULTY }

procedure DISPLAY_COMMITTEE;
begin { DISPLAY_COMMITTEE }
   COUNT := 0;
   if C_INST then
       begin
           COMMITTEE_QUERY := P;
           writeln(FSQL,
             *************************
             ***************************':60);
           write  (FSQL,
           '*':10,
           COMMITTEE_QUERY.OBJECT_INFO.OBJECT_NAME:25);
           writeln(FSQL,
           ' QUERY ','*':18);
           writeln(FSQL,
           '***************************
            *************************':60);
       end;
       writeln('*************************
                ***************************':60);
       write  ('*':10,
       COMMITTEE_QUERY.OBJECT_INFO.OBJECT_NAME:25);
       writeln(' QUERY ','*':18);
       writeln('*************************
                ***************************':60);
```

```
      if C_INST then
         write (FSQL,
         '*':10,
         COMMITTEE_QUERY.OBJECT_INFO.OBJECT_NAME:16,
         '*':10);
         write ('*':10,
         COMMITTEE_QUERY.OBJECT_INFO.OBJECT_NAME:16,
         '*':10);
         DISPLAY_OBJECT_INSTRUCTIONS( C_INST,
         COMMITTEE_QUERY );
      if C_INST then
         write (FSQL,
        '*':10,
        COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[1]:7,
        '*':19);
        write ('*':10,
        COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[1]:7,
        '*':19);

        DISPLAY_ATTRIBUTE_OPERATIONS( C_INST,
        COMMITTEE_QUERY );
        if C_INST then
        write (FSQL,
        '*':10,
        COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[2]:9,
        '*':17);
        write ('*':10,
        COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[2]:9,
        '*':17);
        DISPLAY_ATTRIBUTE_OPERATIONS( C_INST,
        COMMITTEE_QUERY );
   if C_INST then
        write (FSQL,
        '*':10,
        COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[3]:9,
        '*':17);
        write ('*':10,
        COMMITTEE_QUERY.ATTRIBUTES.ATTR_NAME[3]:9,
        '*':17);
        DISPLAY_ATTRIBUTE_OPERATIONS( C_INST,
        COMMITTEE_QUERY );
   if C_INST then
   begin
       writeln(FSQL,
       '************************
        ***************************':60);
       writeln(FSQL);
   end;
   writeln('*************************
             ***************************':60);
   writeln;
   if C_INST then begin
        SUBSCRIPT := SUBSCRIPT + 1;
        QUERY[SUBSCRIPT] := COMMITTEE_QUERY;
        QUERY[SUBSCRIPT].IS_QUERIED := true;
    end;  { if C_INST }

   CLEAR(C_INST);
   P := COMMITTEE_QUERY;
 end; { DISPLAY_COMMITTEE }


procedure GET_INSTRUCTIONS (Var P: PARTIAL_QUERY);
 var
     w: integer;
 begin { GET_INSTRUCTIONS }
     write ('INPUT INSTRUCTIONS FOR OBJECT ');
     write (P.OBJECT_INFO.OBJECT_NAME, ':  ');
     readln ( P.OBJECT_INFO.OBJECT_INST);
     for w := 1 to 3 do begin
         write ('INPUT INSTRUCTIONS FOR ATTRIBUTE ',
                 P.ATTRIBUTES.ATTR_NAME[w], ':  ');
         readln (P.ATTRIBUTES.ATTR_OPERATION[w]);
     end; { for w := 1 to 3 }

     if (P.OBJECT_INFO.OBJECT_NAME = 'EMPLOYEE')  or
        (P.OBJECT_INFO.OBJECT_NAME = 'STUDENT') then
     begin
         write ('INPUT INSTRUCTIONS FOR ATTRIBUTE ',
                P.ATTRIBUTES.ATTR_NAME[4], ':  ');
         readln (P.ATTRIBUTES.ATTR_OPERATION[4]);
     end; { P NAME = 'EMPLOYEE or STUDENT' }

     if (P.OBJECT_INFO.OBJECT_NAME = 'STUDENT') then
     begin
         write ('INPUT INSTRUCTIONS FOR ATTRIBUTE ',
                P.ATTRIBUTES.ATTR_NAME[5], ':  ');
         readln (P.ATTRIBUTES.ATTR_OPERATION[5]);
         write ('INPUT INSTRUCTIONS FOR ATTRIBUTE ',
                P.ATTRIBUTES.ATTR_NAME[6], ':  ');
         readln (P.ATTRIBUTES.ATTR_OPERATION[6]);
         if P.ATTRIBUTES.ATTR_OPERATION[6] <> '' then
             MAJOR := true;
     end; { P = STUDENT_QUERY }

     if (P.OBJECT_INFO.OBJECT_NAME = 'COMMITTEE') then
        if P.ATTRIBUTES.ATTR_OPERATION[2] <> '' then
             MEMBERS := true;
        if (P.OBJECT_INFO.OBJECT_NAME = 'FACULTY') then
        if P.ATTRIBUTES.ATTR_OPERATION[3] <> '' then
             WORKSFOR := true;
```

```
      if (P.OBJECT_INFO.OBJECT_NAME = 'EMPLOYEE') then
       begin
          if P.ATTRIBUTES.ATTR_OPERATION[3] <> '' then
                DEPT := true;
          if P.ATTRIBUTES.ATTR_OPERATION[4] <> '' then
                JOBTYPE := true;
       end;

    clrscr;

end; { GET_INSTRUCTIONS }



procedure SHOW_USER_HIS_ENTRY;
begin { SHOW_USER_HIS_ENTRY }
    case OPTION of
        's','S': begin
                     S_INST := true;
                     STUDENT_QUERY.IS_QUERIED :=
                     true;
                     DISPLAY_STUDENT;
                     S_INST := false;
                 end;
        'e','E': begin
                     E_INST := true;
                     EMPLOYEE_QUERY.IS_QUERIED :=
                     true;
                     DISPLAY_EMPLOYEE;
                     E_INST := false;
                 end;
        'f','F': begin
                     F_INST := true;
                     FACULTY_QUERY.IS_QUERIED :=
                     true;
                     DISPLAY_FACULTY;
                     F_INST := false;
                 end;
        'c','C': begin
                     C_INST := true;
                     COMMITTEE_QUERY.IS_QUERIED :=
                     true;
                     DISPLAY_COMMITTEE;
                     C_INST := false;
                   end;
    end; { case OPTION of }

end;  { SHOW_USER_HIS_ENTRY }
```

```
begin     { DISPLAY_QUERY_OPTIONS }

    SUBSCRIPT := 0;
    S_INST := false; E_INST := false;
    F_INST := false; C_INST := false;
    clrscr;
    for V := 1 to 5 do writeln;
    while ( OPTION <> 'Q' ) and ( OPTION <> 'q' ) do
     begin
        writeln ('ENTER "S" FOR STUDENT QUERY':51);
        writeln;
        writeln ('ENTER "E" FOR EMPLOYEE QUERY':52);
        writeln;
        writeln ('ENTER "F" FOR FACULTY QUERY':51);
        writeln;
        writeln ('ENTER "C" FOR COMMITTEE QUERY':53);
        writeln;
        writeln ('ENTER "Q" TO QUIT SELECTION':51);
        writeln;
        write (' ':31);
        readln (OPTION);
        writeln;
        clrscr;

        case OPTION of
            's','S': DISPLAY_STUDENT;
            'e','E': DISPLAY_EMPLOYEE;
            'f','F': DISPLAY_FACULTY;
            'c','C': DISPLAY_COMMITTEE;
        end; { case OPTION of }

        if ( OPTION <> 'Q') and ( OPTION <> 'q' ) then
        begin
            GET_INSTRUCTIONS(P);
            SHOW_USER_HIS_ENTRY;
        end; { if OPTION }
            for V := 1 to 5 do writeln;

    end; {   while OPTION <> 'q' and 'Q' }

end;     { DISPLAY_QUERY_OPTIONS }


procedure HOW_MANY_PARTIAL_QUERIES( Var PARTIALS:
                                        integer);
begin     { HOW_MANY_PARTIAL_QUERIES }
    PARTIALS := 0;
    while QUERY[PARTIALS + 1].IS_QUERIED do
        PARTIALS := PARTIALS + 1;
end;     { HOW_MANY_PARTIAL_QUERIES }
```

```
function MORE_THAN_ONE_OBJECT_USED_IN_QUERY: boolean;
var
    i: integer;
begin    { MORE_THAN_ONE_OBJECT_USED_IN_QUERY }
    i := 0;
    while QUERY[i + 1].IS_QUERIED do
        i := i + 1;
        MORE_THAN_ONE_OBJECT_USED_IN_QUERY := ( 1 < i );
end;     { MORE_THAN_ONE_OBJECT_USED_IN_QUERY }


procedure SELECT(k: integer);
{ Analogous to Relational SELECT
  Substituted for Object .P }
begin     { SELECT }
    ERSQL[k] := 'SELECT ';
end;      { SELECT }


procedure FROM(j,k: integer);
{ Analgous to Relational FROM - Object Name }
begin      { FROM }
    ERSQL[k] := 'FROM   ' +
    QUERY[j].OBJECT_INFO.OBJECT_NAME;
end;      { FROM }


procedure JOIN_OBJECTS;
{ Unless * is used for attribute name, the selection
  of more than one object window implicitly specifies
  a join of Tables }
begin     { JOIN }
    case NO_PARTIAL_QUERIES_IN_QUERY of
       2: ERSQL[2] := ERSQL[2]  +  ', '  +
          QUERY[2].OBJECT_INFO.OBJECT_NAME;
       3: ERSQL[2] := ERSQL[2]  +  ', '  +
          QUERY[2].OBJECT_INFO.OBJECT_NAME
          +  ', '  +  QUERY[3].OBJECT_INFO.OBJECT_NAME;
       4: ERSQL[2] := ERSQL[2]  +  ', '  +
          QUERY[2].OBJECT_INFO.OBJECT_NAME
          +  ', '  +  QUERY[3].OBJECT_INFO.OBJECT_NAME
          +  ', '  +  QUERY[4].OBJECT_INFO.OBJECT_NAME;
    end; { case NUM_PARTIAL_QUERIES_IN_QUERY of }
    JOIN := true;
end;     { JOIN }


function INSTRUCTIONS_ON_OBJECT_NAME(i:integer):boolean;
begin     { INSTRUCTIONS_ON_OBJECT_NAME }
    INSTRUCTIONS_ON_OBJECT_NAME :=
        ' ' <> QUERY[i].OBJECT_INFO.OBJECT_INST[1];
end;     { INSTRUCTIONS_ON_OBJECT_NAME }
```

```
procedure OBJ_OPERATIONS(i: integer);
begin     { OBJ_OPERATIONS }
    if  ( ERSQL[1] = 'SELECT ')  and
        ( QUERY[i].OBJECT_INFO.OBJECT_INST = '*' ) then
        ERSQL[1] :=  ERSQL[1] +
        QUERY[i].OBJECT_INFO.OBJECT_NAME + '.' + '*'
    else if  ( ERSQL[1] <> 'SELECT ')  and
        ( QUERY[i].OBJECT_INFO.OBJECT_INST = '*' ) then
         ERSQL[1] :=  ERSQL[1] + ', ' +
         QUERY[i].OBJECT_INFO.OBJECT_NAME + '.' + '*'
    else if  ( ERSQL[1] = 'SELECT ')  and
        ( QUERY[i].OBJECT_INFO.OBJECT_INST = 'COUNT' )
        then ERSQL[1] :=  ERSQL[1] + 'COUNT(*)'
    else if  ( ERSQL[1] <> 'SELECT ')  and
        ( QUERY[i].OBJECT_INFO.OBJECT_INST = 'COUNT' )
        then ERSQL[1] :=  ERSQL[1] + ', ' +  'COUNT(*)';
end;     { OBJ_OPERATIONS }


function EXECUTE_AND_DECODE(i: integer): boolean;
{ Global counter X is used to step through all of
  the attributes until the query window is decoded
   and translated to ERSQL }
begin     { EXECUTE_AND_DECODE }
    if QUERY[i].OBJECT_INFO.OBJECT_NAME = 'STUDENT'
    then EXECUTE_AND_DECODE := ( X < 7 )
    else if QUERY[i].OBJECT_INFO.OBJECT_NAME =
    'EMPLOYEE' then
        EXECUTE_AND_DECODE := ( X < 5 )
    else EXECUTE_AND_DECODE := ( X < 4 );
end;     { EXECUTE_AND_DECODE }


function ASTERISK_FOR_ATTRIBUTE_NAME(i:integer):
                                       boolean;
{ Selection of * attribute indicated a nested query }
begin     { ASTERISK_FOR_ATTRIBUTE_NAME }
    ASTERISK_FOR_ATTRIBUTE_NAME :=
        QUERY[i].ATTRIBUTES.ATTR_NAME[X] = '*';
end;     { ASTERISK_FOR_ATTRIBUTE_NAME }


function EXISTENTIAL_QUANTIFIER(i: integer): boolean;
{ Used with nested queries }
begin     { EXISTENTIAL_QUANTIFIER }
    EXISTENTIAL_QUANTIFIER :=
    (QUERY[i].ATTRIBUTES.ATTR_OPERATION[X] = 'EXISTS')
    or (QUERY[i].ATTRIBUTES.ATTR_OPERATION[X] =
                                         'NOT EXISTS');
end;     { EXISTENTIAL_QUANTIFIER }
```

```
function DOT_P_ON_ATTRIBUTE(i:integer):boolean;
{ Indicates print or .P = SELECT attribute }
var
   TEMP: string[30];
begin     { DOT_P_ON_ATTRIBUTE }
    TEMP := '';
    TEMP := QUERY[i].ATTRIBUTES.ATTR_OPERATION[X];
    DOT_P_ON_ATTRIBUTE := ( (TEMP[1] = '.') and
    (TEMP[2] = 'P') );
end;     { DOT_P_ON_ATTRIBUTE }

procedure RETRIEVE_SELECT(i: integer);
{ Checks to see what is selected }
    procedure CHECK_OPS (i: integer; COMPARE: NAME);
    { Allows attribute with mathematical operation
      to selected DISTINCT, COUNT, or other
      AGGREGATE Ops }
    begin { CHECK_OPS }
        if ( COMPARE[1] in ['+', '-', '*', '/']  ) then
             ERSQL[1] := ERSQL[1] +
             QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                            +  COMPARE
            else  if COMPARE = '' then
                ERSQL[1] := ERSQL[1] +
                QUERY[i].ATTRIBUTES.ATTR_NAME[X]
            else  if COMPARE = 'DISTINCT' then
                ERSQL[1] := ERSQL[1] + 'DISTINCT '
                +   QUERY[i].ATTRIBUTES.ATTR_NAME[X]
            else  if COMPARE = 'COUNT' then
                ERSQL[1] := ERSQL[1] + 'COUNT('
                +    QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                +  ')'
            else  if COMPARE = 'SUM'   then
                ERSQL[1] := ERSQL[1] + 'SUM('
                +    QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                +  ')'
            else  if COMPARE = 'AVG'   then
                ERSQL[1] := ERSQL[1] + 'AVG('
                +    QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                +  ')'
            else  if COMPARE = 'MAX'   then
                ERSQL[1] := ERSQL[1] + 'MAX('
                +    QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                +  ')'
            else  if COMPARE = 'MIN'   then
                ERSQL[1] := ERSQL[1] + 'MIN('
                +    QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                +  ')';
    end; { CHECK_OPS }
```

```
var
    COMPARE: NAME;

begin     { RETRIEVE_SELECT }
    COMPARE := '';
    COMPARE :=
    copy(QUERY[i].ATTRIBUTES.ATTR_OPERATION[X],4,27);
    if  ( MORE_THAN ) or
       ( INSTRUCTIONS_ON_OBJECT_NAME(i) ) then
        begin { if MORE_THAN }
            ERSQL[1] := ERSQL[1] + ', ';
            CHECK_OPS (i, COMPARE);
        end { if MORE_THAN }
    else if not MORE_THAN then
    begin { if not MORE_THAN }
            CHECK_OPS (i, COMPARE);
        end; { if not MORE_THAN }
    MORE_THAN := true; X := X + 1;
end;     { RETRIEVE_SELECT }


function NO_INSTRUCTIONS(i: integer): boolean;
begin      { NO_INSTRUCTIONS }
    NO_INSTRUCTIONS :=
       QUERY[i].ATTRIBUTES.ATTR_OPERATION[X] = '';
end;     { NO_INSTRUCTIONS }


procedure NO_OP( i: integer);
begin     { NO_OP }
    { DOES NOTHING BUT IS USED TO ILLUSTRATE THE MAIN }
    { LOOP'S ALGORITHM FOR DECODING THE OBJECT QUERY  }
    X := X + 1;
end;    { NO_OP }


function INSTRUCTIONS_ON_ATTRIBUTE(i: integer): boolean;
begin     { INSTRUCTIONS_ON_ATTRIBUTE }
    INSTRUCTIONS_ON_ATTRIBUTE :=
        QUERY[i].ATTRIBUTES.ATTR_OPERATION[X] <> '';
end;     { INSTRUCTIONS_ON_ATTRIBUTE }


procedure RELATIONAL_OPERATORS;
var
    TEMP: string[30];
begin     { RELATIONAL_OPERATORS }
TEMP := QUERY[i].ATTRIBUTES.ATTR_OPERATION[X];
```

```
if TEMP[1] in ['>','<','='] then
    begin { if TEMP[1] }
        if ( ERSQL[e+2] = SENTINEL ) then
            begin { ERSQL[e+2] =   SENTINEL }
                ERSQL[e+2] := 'WHERE  '
                   + QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                   +  ' ' +  TEMP;
            end   { ERSQL[e+2] = SENTINEL }

        else
            begin { ERSQL[e+2] <> SENTINEL }
                ERSQL[e+3] := 'AND    '
                   +  QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                   +  ' ' +  TEMP;
                e := e + 1;
            end;  { ERSQL[e+2] <> SENTINEL }
    end  { if TEMP[1] }
else
    begin { else }
        if ( ERSQL[e+2] = SENTINEL ) then
            begin { ERSQL[e+2] =   SENTINEL }
                ERSQL[e+2] := 'WHERE  '
                   + QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                   + ' SYNTAX ERROR W/REL OP';
                e := e + 3;
            end   { ERSQL[e+2] = SENTINEL }
         else
            begin { ERSQL[e+2] }
                ERSQL[e] := 'AND    '
                +  QUERY[i].ATTRIBUTES.ATTR_NAME[X]
                +  ' SYNTAX ERROR W/REL OP';
                e := e + 1;
            end;  { ERSQL[e+2] <> SENTINEL }
    end;  { else }  X := X + 1;
end;     { RELATIONAL_OPERATORS }


procedure SPECIAL_OPS(i: integer);
begin     { SPECIAL_OPS }
    e := 3;
    ERSQL[2] := 'FROM   ' +
    QUERY[1].OBJECT_INFO.OBJECT_NAME;
    if QUERY[i].ATTRIBUTES.ATTR_OPERATION[X] = 'EXISTS'
    then ERSQL[e] := 'WHERE  EXISTS';
    if QUERY[i].ATTRIBUTES.ATTR_OPERATION[X] =
    'NOT EXISTS' then
        ERSQL[e] := 'WHERE  NOT EXISTS';
    e := e + 1;
    ERSQL[e] := '( SELECT *';
    FROM(i,e+1); X := X + 1;
end;     { SPECIAL_OPS }
```

```
procedure ESQL_VIEW;
var
    i: integer;
    LEFT_PAREN: boolean;
    TEMP: string[50];
begin     { ESQL_VIEW }
    i := 1;
    LEFT_PAREN := false;
    while ERSQL[i] <> SENTINEL do
        begin { ERSQL[i] <> SENTINEL }
            TEMP := ERSQL[i];
            if (TEMP[1] = '(') then
               LEFT_PAREN := true;
            if LEFT_PAREN then
                begin { LEFT_PAREN }
                    if TEMP[1] = '(' then
                        ERSQL[i] := '     '
                        + TEMP
                    else
                        ERSQL[i] := '       ' + TEMP;
                end;  { LEFT_PAREN }
            if (ERSQL[i+1] = SENTINEL) and
               LEFT_PAREN then
                ERSQL[i] := ERSQL[i] + ')';
            writeln(ERSQL[i]);
            i := i + 1;
        end;  { ERSQL[i] <> SENTINEL }
end;     { ESQL_VIEW }


procedure SQL_VIEW;
var
    r: char;
    i: integer;
    j: DEPT_SET;
    k: JOBS;
    CONTINUE, DONE, SETTABLE1,
    SETTABLE2, SETTABLE3, SETTABLE4: boolean;

    procedure TRANS_MAJOR(VAR CONTINUE, DONE, SETTABLE1,
                          SETTABLE2,SETTABLE3, SETTABLE4:
                  boolean; VAR i:integer;
                          j: DEPT_SET; k:JOBS);
  begin { TRAN_MAJOR }
      if ERSQL[i] = 'WHERE  MAJOR = "COMP.SCI."' then
         SQL[i] := 'WHERE  IDNAME = "COMP.SCI."'
      else if ERSQL[i] = 'WHERE  MAJOR <> "COMP.SCI."'
      then SQL[i] := 'WHERE  DIDNAME <> "COMP.SCI."'
      else if ERSQL[i] = 'AND    MAJOR = "COMP.SCI."'
      then SQL[i] := 'AND    DIDNAME = "COMP.SCI."'
      else if ERSQL[i] = 'AND    MAJOR <> "COMP.SCI."'
      then SQL[i] := 'AND    DIDNAME <> "COMP.SCI."'
    else if ERSQL[i] = 'WHERE  MAJOR = "MATH"' then
    SQL[i] := 'WHERE  DIDNAME = "MATH"'
    else if ERSQL[i] = 'WHERE  MAJOR <> "MATH"' then
    SQL[i] := 'WHERE  DIDNAME <> "MATH"'
    else if ERSQL[i] = 'AND    MAJOR = "MATH"' then
    SQL[i] := 'AND    DIDNAME = "MATH"'
    else if ERSQL[i] = 'AND    MAJOR <> "MATH"' then
    SQL[i] := 'AND    DIDNAME <> "MATH"'
    else if ERSQL[i] = 'WHERE  MAJOR = "MUSIC"' then
    SQL[i] := 'WHERE  DIDNAME = "MUSIC"'
    else if ERSQL[i] = 'WHERE  MAJOR <> "MUSIC"' then
    SQL[i] := 'WHERE  DIDNAME <> "MUSIC"'
    else if ERSQL[i] = 'AND    MAJOR = "MUSIC"' then
    SQL[i] := 'AND    DIDNAME = "MUSIC"'
    else if ERSQL[i] = 'AND    MAJOR <> "MUSIC"' then
    SQL[i] := 'AND    DIDNAME <> "MUSIC"'
    else if ERSQL[i] = 'WHERE  MAJOR <> "PHYSICS"'
    then SQL[i] := 'WHERE  DIDNAME <> "PHYSICS"'
    else if ERSQL[i] = 'AND    MAJOR = "PHYSICS"'
    then SQL[i] := 'AND    DIDNAME = "PHYSICS"'
    else if ERSQL[i] = 'AND    MAJOR <> "PHYSICS"'
    then SQL[i] := 'AND    DIDNAME <> "PHYSICS"'
    else if ERSQL[i] = 'WHERE  MAJOR = "PHYSICS"'
    then SQL[i] := 'WHERE  DIDNAME = "PHYSICS"'
    else if ERSQL[i] = 'WHERE  MAJOR <> "HISTORY"'
    then SQL[i] := 'WHERE  DIDNAME <> "HISTORY"'
    else if ERSQL[i] = 'AND    MAJOR = "HISTORY"'
    then SQL[i] := 'AND    DIDNAME = "HISTORY"'
    else if ERSQL[i] = 'AND    MAJOR <> "HISTORY"'
    then SQL[i] := 'AND    DIDNAME <> "HISTORY"'
    else if ERSQL[i] = 'WHERE  MAJOR = "HISTORY"'
    then SQL[i] := 'WHERE  DIDNAME = "HISTORY"'
    else CONTINUE := TRUE;

    if (not SETTABLE1) and (not SETTABLE2) and
    (not SETTABLE3) and (not SETTABLE4) and
    (i = 2) then
    begin  { SETTABLE }
        SQL[2] := ERSQL[2] + ', IDDEPT';
        SETTABLE1 := true;
    end   { SETTABLE }
    else if (not SETTABLE1) and (i > 2) then
    begin  { SETTABLE }
    SQL[2] := SQL[2] + ', IDDEPT';
        SETTABLE1 := true;
    end;   { SETTABLE }
    if not CONTINUE then begin
    MAJOR := FALSE;
    DONE := true;
    end;
end;  { TRAN_MAJOR }
```

```
procedure TRANS_DEPT(VAR CONTINUE, DONE, SETTABLE1,
                         SETTABLE2, SETTABLE3,
                         SETTABLE4: boolean; VAR i:
                         integer; j: DEPT_SET; k:JOBS);
begin { TRANS_DEPT }
    if ERSQL[i] = 'WHERE  DEPT = "COMP.SCI."' then
    SQL[i] := 'WHERE  DIDNAME = "COMP.SCI."'
    else if ERSQL[i] = 'WHERE  DEPT <> "COMP.SCI."'
    then SQL[i] := 'WHERE  DIDNAME <> "COMP.SCI."'
    else if ERSQL[i] = 'AND    DEPT = "COMP.SCI."'
    then SQL[i] := 'AND    DIDNAME = "COMP.SCI."'
    else if ERSQL[i] = 'AND    DEPT <> "COMP.SCI."'
    then SQL[i] := 'AND    DIDNAME <> "COMP.SCI."'
    else if ERSQL[i] = 'WHERE  DEPT = "MATH"' then
    SQL[i] := 'WHERE  DIDNAME = "MATH"'
    else if ERSQL[i] = 'WHERE  DEPT <> "MATH"' then
    SQL[i] := 'WHERE  DIDNAME <> "MATH"'
    else if ERSQL[i] = 'AND    DEPT = "MATH"' then
    SQL[i] := 'AND    DIDNAME = "MATH"'
    else if ERSQL[i] = 'AND    DEPT <> "MATH"' then
    SQL[i] := 'AND    DIDNAME <> "MATH"'
    else if ERSQL[i] = 'WHERE  DEPT = "MUSIC"' then
    SQL[i] := 'WHERE  DIDNAME = "MUSIC"'
    else if ERSQL[i] = 'WHERE  DEPT <> "MUSIC"' then
    SQL[i] := 'WHERE  DIDNAME <> "MUSIC"'
    else if ERSQL[i] = 'AND    DEPT = "MUSIC"' then
    SQL[i] := 'AND    DIDNAME = "MUSIC"'
    else if ERSQL[i] = 'AND    DEPT <> "MUSIC"' then
    SQL[i] := 'AND    DIDNAME <> "MUSIC"'
    else if ERSQL[i] = 'WHERE  DEPT <> "PHYSICS"'
    then SQL[i] := 'WHERE  DIDNAME <> "PHYSICS"'
    else if ERSQL[i] = 'AND    DEPT = "PHYSICS"'
    then SQL[i] := 'AND    DIDNAME = "PHYSICS"'
    else if ERSQL[i] = 'AND    DEPT <> "PHYSICS"'
    then SQL[i] := 'AND    DIDNAME <> "PHYSICS"'
    else if ERSQL[i] = 'WHERE  DEPT = "PHYSICS"'
    then SQL[i] := 'WHERE  DIDNAME = "PHYSICS"'
    else if ERSQL[i] = 'WHERE  DEPT <> "HISTORY"'
    then SQL[i] := 'WHERE  DIDNAME <> "HISTORY"'
    else if ERSQL[i] = 'AND    DEPT = "HISTORY"'
    then SQL[i] := 'AND    DIDNAME = "HISTORY"'
    else if ERSQL[i] = 'AND    DEPT <> "HISTORY"'
    then SQL[i] := 'AND    DIDNAME <> "HISTORY"'
    else if ERSQL[i] = 'WHERE  DEPT = "HISTORY"'
    then SQL[i] := 'WHERE  DIDNAME = "HISTORY"'
    else CONTINUE := true;
    if (not SETTABLE1) and  (not SETTABLE2) and
    (not SETTABLE3) and (not SETTABLE4) and (i = 2)
    then begin  { SETTABLE }
       SQL[2] := ERSQL[2] + ', IDDEPT';
       SETTABLE1 := true;
    end   { SETTABLE }
       else if (not SETTABLE1) and (i > 2) then
              begin  { SETTABLE }
                  SQL[2] := SQL[2] + ', IDDEPT';
                  SETTABLE1 := true;
              end;   { SETTABLE }
          if not CONTINUE then begin
             DEPT := FALSE;
             DONE := true;
          end;
end;  { TRANS_DEPT }

procedure TRANS_JOBTYPE(VAR CONTINUE,DONE,SETTABLE1,
                        SETTABLE2,SETTABLE3,
                        SETTABLE4: boolean;
                        VAR i: integer; j: DEPT_SET;
                        k:JOBS);
begin { TRANS_JOBTYPE }
    if ERSQL[i] = 'WHERE  JobType = "FACULTY"'
    then SQL[i] := 'WHERE  EIDNAME = "FACULTY"'
    else if ERSQL[i] = 'WHERE  JobType <> "FACULTY"'
    then SQL[i] := 'WHERE  EIDNAME <> "FACULTY"'
    else if ERSQL[i] = 'AND    JobType = "FACULTY"'
    then SQL[i] := 'AND    EIDNAME = "FACULTY"'
    else if ERSQL[i] = 'AND    JobType <> "FACULTY"'
    then SQL[i] := 'AND    EIDNAME <> "FACULTY"'
    else if ERSQL[i] = 'WHERE  JobType = "SECRETARY"'
    then SQL[i] := 'WHERE  EIDNAME = "SECRETARY"'
    else if ERSQL[i] = 'WHERE  JobType <> "SECRETARY"'
    then SQL[i] := 'WHERE  EIDNAME <> "SECRETARY"'
    else if ERSQL[i] = 'AND    JobType = "SECRETARY"'
    then SQL[i] := 'AND    EIDNAME = "SECRETARY"'
    else if ERSQL[i] = 'AND    JobType <> "SECRETARY"'
    then SQL[i] := 'AND    EIDNAME <> "SECRETARY"'
    else CONTINUE := true;

    if (not SETTABLE2) and  (not SETTABLE1) and
    (not SETTABLE3) and (not SETTABLE4) and (i = 2)
    then     begin  { SETTABLE }
                 SQL[2] := ERSQL[2] + ', IDEMP';
                 SETTABLE2 := true;
             end   { SETTABLE }
          else if (not SETTABLE2) and (i > 2) then
              begin  { SETTABLE }
                  SQL[2] := SQL[2] + ', IDEMP';
                  SETTABLE2 := true;
              end;   { SETTABLE }


          if not CONTINUE then begin
             JOBTYPE := FALSE;
             DONE := true;
          end;
end;  { TRANS_JOBTYPE }

procedure TRANS_WORKSFOR(VAR CONTINUE,DONE, SETTABLE1,
                             SETTABLE2, SETTABLE3,
                             SETTABLE4: boolean;
                         VAR i: integer; j: DEPT_SET;
                             k:JOBS);
begin { TRANS_WORKSFOR }
    if ERSQL[i] = 'WHERE  WorksFor = "COMP.SCI."' then
       SQL[i] := 'WHERE  DIDNAME = "COMP.SCI."'
    else if ERSQL[i] = 'WHERE  WorksFor <> "COMP.SCI."'
         then SQL[i] := 'WHERE  DIDNAME <> "COMP.SCI."'
    else if ERSQL[i] = 'AND    WorksFor = "COMP.SCI."'
         then SQL[i] := 'AND    DIDNAME = "COMP.SCI."'
    else if ERSQL[i] = 'AND    WorksFor <> "COMP.SCI."'
         then SQL[i] := 'AND    DIDNAME <> "COMP.SCI."'
    else if ERSQL[i] = 'WHERE  WorksFor = "MATH"' then
          SQL[i] := 'WHERE  DIDNAME = "MATH"'
    else if ERSQL[i] = 'WHERE  WorksFor <> "MATH"' then
          SQL[i] := 'WHERE  DIDNAME <> "MATH"'
    else if ERSQL[i] = 'AND    WorksFor = "MATH"' then
          SQL[i] := 'AND    DIDNAME = "MATH"'
    else if ERSQL[i] = 'AND    WorksFor <> "MATH"' then
          SQL[i] := 'AND    DIDNAME <> "MATH"'
    else if ERSQL[i] = 'WHERE  WorksFor = "MUSIC"' then
              SQL[i] := 'WHERE  DIDNAME = "MUSIC"'
    else if ERSQL[i] = 'WHERE  WorksFor <> "MUSIC"' then
              SQL[i] := 'WHERE  DIDNAME <> "MUSIC"'
    else if ERSQL[i] = 'AND    WorksFor = "MUSIC"' then
              SQL[i] := 'AND    DIDNAME = "MUSIC"'
    else if ERSQL[i] = 'AND    WorksFor <> "MUSIC"' then
              SQL[i] := 'AND    DIDNAME <> "MUSIC"'
    else if ERSQL[i] = 'WHERE  WorksFor <> "PHYSICS"'
         then SQL[i] := 'WHERE  DIDNAME <> "PHYSICS"'
    else if ERSQL[i] = 'AND    WorksFor = "PHYSICS"'
         then SQL[i] := 'AND    DIDNAME = "PHYSICS"'
    else if ERSQL[i] = 'AND    WorksFor <> "PHYSICS"'
         then SQL[i] := 'AND    DIDNAME <> "PHYSICS"'
    else if ERSQL[i] = 'WHERE  WorksFor = "PHYSICS"'
         then SQL[i] := 'WHERE  DIDNAME = "PHYSICS"'
    else if ERSQL[i] = 'WHERE  WorksFor <> "HISTORY"'
```

```
         then SQL[i] := 'WHERE  DIDNAME <> "HISTORY"'
    else if ERSQL[i] = 'AND    WorksFor = "HISTORY"'
         then SQL[i] := 'AND    DIDNAME = "HISTORY"'
    else if ERSQL[i] = 'AND    WorksFor <> "HISTORY"'
         then SQL[i] := 'AND    DIDNAME <> "HISTORY"'
    else if ERSQL[i] = 'WHERE  WorksFor = "HISTORY"'
         then SQL[i] := 'WHERE  DIDNAME = "HISTORY"'
    else CONTINUE := true;

    if (not SETTABLE1) and  (not SETTABLE2) and
        (not SETTABLE3) and (not SETTABLE4) and
        (i = 2) then
        begin  { SETTABLE }
            SQL[2] := ERSQL[2] + ', IDDEPT';
            SETTABLE1 := true;
        end   { SETTABLE }
    else if (not SETTABLE1) and (i > 2) then
        begin  { SETTABLE }
            SQL[2] := SQL[2] + ', IDDEPT';
            SETTABLE1 := true;
        end;   { SETTABLE }
    if not CONTINUE then begin
        DEPT := FALSE;
        DONE := true;
    end;

end; { TRANS_WORKSFOR }



procedure TRANS_MEMBERS(VAR CONTINUE,DONE,SETTABLE1,
                            SETTABLE2, SETTABLE3,
                            SETTABLE4: boolean;
                        VAR i:
                        integer; j: DEPT_SET; k:JOBS);
begin { TRANS_MEMBERS }
    if ERSQL[i] = 'WHERE  MEMBERS = LUM'
    then SQL[i] := 'WHERE  FIDNAME = "LUM"'
    else if ERSQL[i] = 'WHERE  MEMBERS <> "LUM"'
    then SQL[i] := 'WHERE  FIDNAME <> "LUM"'
    else if ERSQL[i] = 'AND    MEMBERS = "LUM"'
    then SQL[i] := 'AND    FIDNAME = "LUM"'
    else if ERSQL[i] = 'AND    MEMBERS <> "LUM"'
    then SQL[i] := 'AND    FIDNAME <> "LUM"'
    else if ERSQL[i] = 'WHERE  MEMBERS = "JOHNSON"'
    then SQL[i] := 'WHERE  FIDNAME = "JOHNSON"'
    else if ERSQL[i] = 'WHERE  MEMBERS <> "JOHNSON"'
    then SQL[i] := 'WHERE  FIDNAME <> "JOHNSON"'
    else if ERSQL[i] = 'AND    MEMBERS = "JOHNSON"'
    then SQL[i] := 'AND    FIDNAME = "JOHNSON"'
    else if ERSQL[i] = 'AND    MEMBERS <> "JOHNSON"'
    then SQL[i] := 'AND    FIDNAME <> "JOHNSON"'
```

```
      else CONTINUE := true;
      if (not SETTABLE2) and  (not SETTABLE1) and
      (not SETTABLE3) and (not SETTABLE4) and (i = 2) then
      begin  { SETTABLE }
          SQL[2] := ERSQL[2] + ', IDCOM';
          SETTABLE3 := true;
      end    { SETTABLE }

      else if (not SETTABLE3) and (i > 2) then
      begin  { SETTABLE }
          SQL[2] := SQL[2] + ', IDCOMP';
          SETTABLE3 := true;
      end;   { SETTABLE }

      if not CONTINUE then begin
          JOBTYPE := FALSE;
          DONE := true;
       end;
end;  { TRANS_MEMBERS }


begin     { SQL_VIEW }
    i := 1;
    writeln;
    writeln;
    write ('HIT ENTER TO VIEW SEQUEL QUERY ':47);
    readln (r);
    ClrScr;
    SETTABLE1 := false; SETTABLE2 := false;
    SETTABLE3 := false;
    SETTABLE4 := false;
    CONTINUE := false;
    DONE := false;
    SQL[1] := ERSQL[1];
    while ERSQL[i] <> SENTINEL do begin
        if MAJOR then
            TRANS_MAJOR( CONTINUE, DONE, SETTABLE1,
                          SETTABLE2, SETTABLE3,
                          SETTABLE4, i, j, k);
        if DEPT and not DONE then
            TRANS_DEPT( CONTINUE, DONE, SETTABLE1,
                         SETTABLE2,
                         SETTABLE3, SETTABLE4, i, j, k);
        if JOBTYPE and not DONE then
            TRANS_JOBTYPE( CONTINUE, DONE, SETTABLE1,
                           SETTABLE2, SETTABLE3,
                          SETTABLE4, i, j, k);
         if WORKSFOR and not DONE then
             TRANS_WORKSFOR( CONTINUE, DONE, SETTABLE1,
                           SETTABLE2,
                         SETTABLE3, SETTABLE4, i, j, k);
        if MEMBERS and not DONE then
            TRANS_MEMBERS( CONTINUE, DONE, SETTABLE1,
                                SETTABLE2,
                             SETTABLE3, SETTABLE4, i, j, k);
                i := i + 1;
                SQL[i] := ERSQL[i];
                DONE := false;
                CONTINUE := false;
    end; { ERSQL[i] <> SENTINEL }


    for i := 1 to 5 do writeln;
    i := 1;
    while SQL[i] <> SENTINEL do
    begin { while SQL[i] <> SENTINEL }
       writeln (SQL[i]);
       i := i + 1;
    end;  { while SQL[i] <> SENTINEL }
end;     { SQL_VIEW }


procedure START_UP;
begin     { START_UP }
    INITIALIZE_GLOBALS;
    LOAD_OBJECTS;
    LOAD_WINDOWS_WITH_OBJECTS;
    DISPLAY_QUERY_OPTIONS;
    HOW_MANY_PARTIAL_QUERIES
        (NO_PARTIAL_QUERIES_IN_QUERY);
end;     { START_UP }

procedure GET_AND_REVIEW_OUTPUT;
var
   i: integer;
begin     { GET_AND_REVIEW_OUTPUT }
    i := 1;
    ESQL_VIEW;
    SQL_VIEW;
    writeln(FSQL); writeln(FSQL);
    writeln(FSQL,'EXTENDED SEQUEL QUERY IS AS FOLLOWS:');
    writeln(FSQL);
    while ERSQL[i] <> SENTINEL do
        begin
             writeln(FSQL, ERSQL[i]);
             i := i + 1;
        end;
    i := 1;
    writeln(FSQL); writeln(FSQL);
    writeln(FSQL,
    'TRANSLATED SEQUEL QUERY IS AS FOLLOWS:');
    writeln(FSQL);
    while SQL[i] <> SENTINEL do
        begin
             writeln(FSQL, SQL[i]);
             i := i + 1;
        end;
    writeln(FSQL); writeln(FSQL);
end;     { GET_AND_REVIEW_OUTPUT }


procedure RESET_GLOBAL_COUNTER;
begin
    X := 1;
end;

begin     { ObjectTranslator }

    FINISHED := false;
    assign (FSQL, 'SEQUEL.SQL');
    rewrite (FSQL);

  REPEAT    { Until all the queries are completed }

    START_UP;
    SELECT(e);
    FROM(e,e+1);
```

```
{***************** TRANSLATION ALGORITHM **************}

    if (MORE_THAN_ONE_OBJECT_USED_IN_QUERY) then
        JOIN_OBJECTS;

    for i := 1 to NO_PARTIAL_QUERIES_IN_QUERY do begin

        if (INSTRUCTIONS_ON_OBJECT_NAME(i)) then
        OBJ_OPERATIONS(i);

        while (EXECUTE_AND_DECODE(i)) do
        begin     { while (EXECUTE_AND_DECODE) }
                  if (ASTERISK_FOR_ATTRIBUTE_NAME(i)) and
                       EXISTENTIAL_QUANTIFIER(i) then
            SPECIAL_OPS(i)
            else if (DOT_P_ON_ATTRIBUTE(i)) then
            RETRIEVE_SELECT(i)
            else if (NO_INSTRUCTIONS(i)) then NO_OP(i)
            else if (INSTRUCTIONS_ON_ATTRIBUTE(i)) then
                RELATIONAL_OPERATORS;
        end;      { while (EXECUTE_AND_DECODE) }

        if (not EXECUTE_AND_DECODE(i)) then
            RESET_GLOBAL_COUNTER;

    end; { for i := 1 to NO_PARITAL_QUERIES_IN_QUERY }

 {*******************************************************}

    GET_AND_REVIEW_OUTPUT;
    writeln; writeln;
    write('ENTER "F" OR "f" TO STOP MAKING QUERIES ':60);
    readln(F);
    if (F = 'F') or (F = 'f') then FINISHED := true;
    UNTIL FINISHED;
    close (FSQL);
end.     { ObjectTranslator }
```

APPENDIX B

TRANSLATOR OUTPUT

```
          ***************************************************
          *                  STUDENT QUERY                  *
          ***************************************************
          *        STUDENT        *                         *
          *     *                 *                         *
          *  SNAME                *               .P        *
          *  ADDRESS              *                         *
          *  SSNO                 *                         *
          *  GPA                  *            >= 3.5       *
          *  MAJOR                *                         *
          ***************************************************

EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT SNAME
FROM   STUDENT
WHERE  GPA >= 3.5


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT SNAME
FROM   STUDENT
WHERE  GPA >= 3.5


          ***************************************************
          *                  FACULTY QUERY                  *
          ***************************************************
          *        FACULTY        *                         *
          *  FNAME                *               .P        *
          *  AGE                  *             >= 30       *
          *  WorksFor             *                         *
          ***************************************************

EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT FNAME
FROM   FACULTY
WHERE  AGE >= 30
```

```
TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT FNAME
FROM   FACULTY
WHERE  AGE >= 30

         ***************************************************
         *                 EMPLOYEE QUERY                  *
         ***************************************************
         *        EMPLOYEE          *                      *
         *  ENAME                   *              .P      *
         *  PAY                     *        >= 30,000     *
         *  DEPT                    *                      *
         *  JobType                 *                      *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT ENAME
FROM   EMPLOYEE
WHERE  PAY >= 30,000


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT ENAME
FROM   EMPLOYEE
WHERE  PAY >= 30,000



         ***************************************************
         *                COMMITTEE QUERY                  *
         ***************************************************
         *       COMMITTEE          *                      *
         *  CNAME                   *              .P      *
         *  MEMBERS                 *                      *
         *  PURPOSE                 *  = "RECRUITING"      *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT CNAME
FROM   COMMITTEE
WHERE  PURPOSE = "RECRUITING"
```

```
TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT CNAME
FROM   COMMITTEE
WHERE  PURPOSE = "RECRUITING"


         ***************************************************
         *                  STUDENT QUERY                  *
         ***************************************************
         *         STUDENT         *                       *
         *     *                   *             COUNT     *
         *  SNAME                  *                       *
         *  ADDRESS                *                       *
         *  SSNO                   *                       *
         *  GPA                    *                       *
         *  MAJOR                  *                       *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT COUNT(*)
FROM   STUDENT


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT COUNT(*)
FROM   STUDENT


         ***************************************************
         *                  STUDENT QUERY                  *
         ***************************************************
         *         STUDENT         *                       *
         *     *                   *             .P        *
         *  SNAME                  *                       *
         *  ADDRESS                *                       *
         *  SSNO                   *                       *
         *  GPA                    *                       *
         *  MAJOR                  *                       *
         ***************************************************

EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT STUDENT.*
FROM   STUDENT
```

```
TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT STUDENT.*
FROM   STUDENT



         ***************************************************
         *                EMPLOYEE QUERY                   *
         ***************************************************
         *        EMPLOYEE         *                       *
         *  ENAME                  *          .P COUNT     *
         *  PAY                    *       > 20,000        *
         *  DEPT                   *                       *
         *  JobType                *                       *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT COUNT(ENAME)
FROM   EMPLOYEE
WHERE  PAY > 20,000


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT COUNT(ENAME)
FROM   EMPLOYEE
WHERE  PAY > 20,000


         ***************************************************
         *                 FACULTY QUERY                   *
         ***************************************************
         *         FACULTY         *                       *
         *  FNAME                  *                       *
         *  AGE                    *            .P         *
         *  WorksFor               *                       *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT AGE
FROM   FACULTY
```

```
TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT AGE
FROM   FACULTY



         ***************************************************
         *                  STUDENT QUERY                  *
         ***************************************************
         *        STUDENT          *                       *
         *     *                   *                       *
         *  SNAME                  *             .P        *
         *  ADDRESS                *                       *
         *  SSNO                   *                       *
         *  GPA                    *          >= 3.0       *
         *  MAJOR                  *                       *
         ***************************************************



         ***************************************************
         *                 EMPLOYEE QUERY                  *
         ***************************************************
         *       EMPLOYEE          *                       *
         *  ENAME                  *           = SNAME     *
         *  PAY                    *        >= 20,000      *
         *  DEPT                   *                       *
         *  JobType                *                       *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:


SELECT SNAME
FROM   STUDENT, EMPLOYEE
WHERE  GPA >= 3.0
AND    ENAME = SNAME
AND    PAY >= 20,000

TRANSLATED SEQUEL QUERY IS AS FOLLOWS:


SELECT SNAME
FROM   STUDENT, EMPLOYEE
WHERE  GPA >= 3.0
AND    ENAME = SNAME
AND    PAY >= 20,000
```

```
**************************************************
*                    EMPLOYEE QUERY              *
**************************************************
*        EMPLOYEE          *                     *
*  ENAME                   *               .P    *
*  PAY                     *        >= 30,000    *
*  DEPT                    *                     *
*  JobType                 *                     *
**************************************************


**************************************************
*                     FACULTY QUERY              *
**************************************************
*         FACULTY          *                     *
*  FNAME                   *           = ENAME   *
*  AGE                     *             >= 30   *
*  WorksFor                *                     *
**************************************************
```

```
EXTENDED SEQUEL QUERY IS AS FOLLOWS:


SELECT ENAME
FROM   EMPLOYEE, FACULTY
WHERE  PAY >= 30,000
AND    FNAME = ENAME
AND    AGE >= 30


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:


SELECT ENAME
FROM   EMPLOYEE, FACULTY
WHERE  PAY >= 30,000
AND    FNAME = ENAME
AND    AGE >= 30
```

```
*****************************************************
*                  FACULTY QUERY                    *
*****************************************************
*         FACULTY          *                        *
*  *                       *                 COUNT  *
*  FNAME                   *                  .P    *
*  AGE                     *       >= AVG(AGE)      *
*  WorksFor                *                        *
*****************************************************


*****************************************************
*                COMMITTEE QUERY                    *
*****************************************************
*       COMMITTEE          *                        *
*  CNAME                   *          = FNAME       *
*  MEMBERS                 *                        *
*  PURPOSE                 *   = "RECRUITING"       *
*****************************************************
```

```
EXTENDED SEQUEL QUERY IS AS FOLLOWS:


SELECT COUNT(*), FNAME
FROM   FACULTY, COMMITTEE
WHERE  AGE >= AVG(AGE)
AND    CNAME = FNAME
AND    PURPOSE = "RECRUITING"


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:


SELECT COUNT(*), FNAME
FROM   FACULTY, COMMITTEE
WHERE  AGE >= AVG(AGE)
AND    CNAME = FNAME
AND    PURPOSE = "RECRUITING"
```

```
         ***************************************************
         *                  STUDENT QUERY                  *
         ***************************************************
         *         STUDENT          *                      *
         *     *                    *                      *
         *  SNAME                   *            .P        *
         *  ADDRESS                 *                      *
         *  SSNO                    *                      *
         *  GPA                     *        .P AVG        *
         *  MAJOR                   *                      *
         ***************************************************

EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT SNAME, AVG(GPA)
FROM   STUDENT


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT SNAME, AVG(GPA)
FROM   STUDENT


         ***************************************************
         *                 EMPLOYEE QUERY                  *
         ***************************************************
         *        EMPLOYEE          *                      *
         *  ENAME                   *                      *
         *  PAY                     *        .P MAX        *
         *  DEPT                    *                      *
         *  JobType                 *   = "SECRETARY"      *
         ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT MAX(PAY)
FROM   EMPLOYEE
WHERE  JobType = "SECRETARY"


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT MAX(PAY)
FROM   EMPLOYEE, IDEMP
WHERE  EIDNAME = "SECRETARY"
```

```
          ***************************************************
          *                  FACULTY QUERY                  *
          ***************************************************
          *        FACULTY        *                         *
          *  FNAME                *                         *
          *  AGE                  *          .P MIN         *
          *  WorksFor             *                         *
          ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT MIN(AGE)
FROM   FACULTY


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT MIN(AGE)
FROM   FACULTY


          ***************************************************
          *                COMMITTEE QUERY                  *
          ***************************************************
          *      COMMITTEE        *                         *
          *  CNAME                *        .P COUNT         *
          *  MEMBERS              *                         *
          *  PURPOSE              *  = "RECRUITING"         *
          ***************************************************


EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT COUNT(CNAME)
FROM   COMMITTEE
WHERE  PURPOSE = "RECRUITING"



TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT COUNT(CNAME)
FROM   COMMITTEE
WHERE  PURPOSE = "RECRUITING"
```

```
***************************************************
*                  STUDENT QUERY                  *
***************************************************
*         STUDENT          *                      *
*     *                    *                      *
*  SNAME                   *              .P      *
*  ADDRESS                 *                      *
*  SSNO                    *                      *
*  GPA                     *          .P MAX      *
*  MAJOR                   *                      *
***************************************************
```

```
EXTENDED SEQUEL QUERY IS AS FOLLOWS:

SELECT SNAME, MAX(GPA)
FROM   STUDENT


TRANSLATED SEQUEL QUERY IS AS FOLLOWS:

SELECT SNAME, MAX(GPA)
FROM   STUDENT
```

** NOTE : With the simulator, * is displayed in window for STUDENT object and only when needed for the query with the other objects.

LIST OF REFERENCES

1. Wu, C.T., "GLAD: Graphics Language for Database," NPS52-87-030, Naval Postgraduate School, July 1987.

2. Zloof, M.M., "Query-by-Example: A Database Language," IBM Systems Journal, Vol, 16, No, 4, 1977.

3. Miyao, Jun'ichi, Design of a User-Friendly Interface for Database Systems, Ph.D. Dissertation, Hiroshima University, Hiroshima, Japan, January 1987.

4. The Whitewater Group, Inc., Actor Language Manual, 1987.

5. Codd, E.F., "A Relational Model for Large Shared Data Banks," Communications of the ACM 13, No. 6 (June 1970), reprinted in Communications of the ACM 26, No. 1 (January 1983).

6. Date, C.J., An Introduction to Database System, Addison Wesley, 1986.

7. Goldman, K.J., Goldman, S.A., Kanellakis, P.C., and Zdonik, S.B., "ISIS: Interface for a Semantic Information System," ACM 0-89791-160-1/85/005/0328, 1985.

8. Bragger, R.P., Dudler, A., Rebsamen, J., Zehnder, C.A., "Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions," IEEE CH2031-3/84/0000/0399$01.00, 1984.

9. Miyao, J., Hirakawa, N., Kikuno, T., Yoshida, N., "Design of a Form Interface Language in Database System Aide," IEEE Workshop on Languages for Automation, 1987.

10. Miyao, J., Tominaga, K., Kikuno, T., and Yoshida, N., "Design of a High Level Query Language for End Users," IEEE Workshop on Languages for Automation, 1986.

11. Korth, H.F., Silberschatz, A., Database System Concepts, McGraw-Hill, 1986.

12. Tsur, S., Zaniolo, C., "An Implementation of GEM--Supporting a Semantic Data Model on a Relational Backend," ACM 0-89791-128-8/84/006/0286, 1984.

13. Chen, P.P.-S., "The Entity-Relationship Model-Toward a Unified View of Data," ACM TODS 1, No. 1 (March 1976).

14. Date, C.J., A Guide to DB2, Addison Wesley, 1984.

15. Wong, K.T., Juo, I., GUIDE: Graphical User Interface for Database Exploration, Applied Mathematic Sciences Research Program of the Office of Energy Research, Department of Energy, 1982.

16. Wu, C.T., Schema and Translation Scheme for GLAD, Thesis Lecture Notes, Naval Postgraduate School, Monterey, California, April 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information System Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Director, Information Systems (OP-945) Office of the Chief of Naval Operations Navy Department Washington, D.C. 20350-2000	1
4.	Curricular Officer Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
5.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
6.	Commandant of the Marine Corps Code TE06 Headquarters, U.S. Marine Corps Washington, D.C. 20360-0001	1
7.	Professor C. Thomas Wu, Code 52Wq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
8.	Capt. Paul D. Grenseman c/o Mr. Hector Licong 4210 Palmira Street Tampa, Florida 33629	5

Thesis
G7656 Grenseman
c.1 Data structures and
algorithms for supporting
GLAD interfaces.